TECHNICAL UNIVERSITY OF CRETE
DEPARTMENT OF ELECTRONIC & COMPUTER ENGINEERING

NeuralStream

A Distributed framework for online pattern discovery in multiple streaming time-series.

Andrew Grammenos

This thesis is submitted in partial fulfillment for the
degree of Electronic & Computer Engineering

Thesis committee
Professor Minos Garofalakis, *Thesis supervisor*
Assistant Professor Vasilis Samoladas
Associate Professor Antonios Deligiannakis

Chania, March 2015

"Never underestimate the attention, risk, money and time that an opponent will put into reading (your) traffic."

The one and only, Robert "Bob" Morris.

Abstract

Developing a *scalable*, *streaming*, *fast* and *flexible* framework for *online* pattern discovery amongst thousands of streams is highly desirable. This is especially evident today since multiple use cases have surfaced for monitoring and analyzing thousands of data-streams in real time while retrieving accurate and reliable metrics from them in a timely manner. In order to scale the computation beyond what a single node can do we have to find a way to split and distribute the vast computational load amongst all the nodes as evenly as possible while maintaining the correctness of our algorithm. As one might think this is not always a trivial task, especially when taking in account the domain of each application.

Through this diploma thesis the author tries to tackle the aforementioned problem by advancing the already existing algorithmic domain. This is achieved by expanding on the already available methods to sufficiently address this problem as well as provide a usable and flexible framework called `NeuralStream` that other users can employ to perform real-time monitoring of thousands of streams. Through `NeuralStream` one can find the representative trends in all of the streams while also offering the flexibility to monitor the streams of each node independently. Thanks to its architecture our framework is extremely robust while also being able to provide processing guarantees in case of multiple node failures; it can also scale extremely well and can optimally split the load given the amount of streams to monitor and the number of available compute nodes in the cluster. The final aggregation of patterns depending on the user preferences is stored and updated by the application, in real-time through a web interface that can be used for displaying charts with the representative trends that exist currently in the monitored streams.

Keywords: Streaming, Online, Distributed Systems, Time-Series, Monitoring, Feature Extraction, Big Data, Distributed Principal Component Analysis, Distributed Projection Approximation Subspace Tracking

Acknowledgements

First off I'd like to thank my thesis supervisor Professor Minos Garofalakis for trusting me and giving me the opportunity to work with him on this intriguing subject. He gave me complete freedom of action and direction while also being there to straighten me up when I deviated from the course we set on and for that I am really indebted to him.

Additionally huge thanks and shout-outs go to Assistant Professor Vasilis Samoladas for always being available to geek out and be part of extremely entertaining conversations with insightful endings. Personal thanks also have to be handed to Associate Professor Antonis Deligiannakis who was always there when I required some feedback or generally guide me through some choices I had to make throughout.

I would not being able to finish my Diploma without the constant support of my parents Antonis and Margarita that wholeheartedly support me for the last 26 and something years; your support means a lot, thank you!

Lastly I'd like to thank my brother for being, well my brother! AS well as the numerous friends that I made while I ventured through this adventure (you know who you are)! The final and most personal thanks must be handed to V. for bearing with me all these years (I know it's not easy) while also constantly motivating me to do more stuff with my life than just write code.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Extracting useful and representative trends from datasets has always been an active topic in scientific research. Useful patterns and trends that are extracted from a number of data-streams find many uses in real world applications...and as one might think, many of them are of particular interest due to their high impact. The applications of such caliber include detecting network anomalies, leakage in sewage pipes, outliers in (distributed) sensor networks and many others.

The main problem that the scientific community has tried to tackle lately is how to scale the detection of such patterns/trends in hundreds or even thousands of streams while maintaining a high quality of detection with feasible computational and storage requirements. This is especially evident in recent years particularly due to the introduction of cloud computing and the rapidly increasing dataset sizes. Also another problem that is of particular interest is how to perform infinite detection of patterns that is required when our dataset does not have a fixed size but instead is ever-expanding in real-time as data arrives (i.e. our sources are *streaming*).

Scaling to accurately monitor thousands of data-streams, extracting the representative patterns within them while providing *processing* and *fail-over* guarantees with all of the challenges involved into materializing that system is, essentially, what motivated me to pursue this thesis.

1.2 Introduction to NeuralStream

This thesis introduces NeuralStream, which is a highly scalable pattern extraction framework that meets the previously mentioned requirements and is built on top of Apache Storm [1]. Apache Storm is an elegant stream processing framework that is ideal for our use-case as it can provide the desirable *processing* and *fail-over* guarantees with ease while also being flexible enough to allow the implementation of our pattern extraction algorithms without any hurdles.

Additionally NeuralStream is able through a web interface to dynamically report a user selectable number of representative trends for the monitored streams in real-time and is designed to run on any cluster that is capable of running Apache Storm without any additional requirements.

1.3 Thesis Contribution

Through this thesis we create a distributed and scalable framework that is able to extract and track the representative trends out of all the monitored data-streams with high quality and fault-tolerance. Additionally it combines the best traits of two algorithms in order to increase its detection effectiveness. This is achieved through firstly performing a streaming and memory limited Principal Component Analysis (PCA) approximation using the algorithm proposed by Mitliagkas et al. in [2] which has very tight theoretical bounds. Then after each of our computation nodes has performed at each step its own approximation we use the techniques outlined by Yang through his Projection Approximation Subspace Tracking (PAST) in [3] to extract k number of projections and "bias" each of node subspaces so that we can make them converge to the global one, thus extracting our representative global patterns out of all our monitored data-streams. To our knowledge this is the first work that attempts to tackle this problem using ideas stemming from these two algorithms and trying to scale the monitoring to thousands of data-streams while also providing strict *processing* and *fail-over* guarantees.

1.4 Thesis overview

This thesis is divided into multiple Chapters; and are 6 in total. Chapter 2 explains the related work that has been done from the scientific community in recent years regarding our topic. Inside Chapter 3 we provide the problem statement and gives a formal definition of the problem we are trying to solve; it also introduces our algorithm and

how we expanded the methods of Mitliagkas et al. [2] and Yang [3] in order to devise it. Chapter 4 describes our implementation and introduces `Apache Storm` stream processor and the reasoning behind its selection. Chapter 5 performs a performance evaluation and genral comparisons against a similar framework such as `D-SPIRIT` in [4]. Finally the closing Chapter 6 discusses the thesis in general and outlines its possible expansion through future work.

Chapter 2

Related Work

This chapter will outline the related work that has been recently performed by the scientific community and that is most closely related with our work, although briefly we will perform an overview of the most commonly used techniques on data-streams. We selected to split this chapter two main sections, the first one involves the related work that has been done for stream approximation in general while the second is related on actual feature extraction (which, in essence, is data-mining) from captured streams.

2.1 Data-Stream Approximation

2.1.1 Data as a stream

Normally our data has a finite and known length, which is given usually by the dataset size as out input; in contrast when we define a data-stream we imply that the size of the actual data length is not known *a-priori*. In our case we consider a data stream as an unbounded data-sequence of consecutive tuples s that each have a $\Delta > 0$ positive and real time-interval between them. Essentially the formal definition of a single-source data-stream is the ordered pair (s, Δ) where s is the tuple sequence and Δ is a real and positive sequence that indicates the time-interval that each tuple has.

Besides the general definition given above, three dominant models have been introduced to describe data-streams; each of which allows or restricts what can be done with current, next and previous values. We will now present these models and describe what they allow to be performed in the data-stream sequence.

- **Time-Series Model**: This is a very computationally friendly model, has vast applications in telecommunications and has been studied extensively. This model

restricts each tuple to update only the current block of the sequence. That essentially means that given a sequence $A[T_1, T_2, ...T_t]$ and a tuple T that arrives at time $t+1$ then this tuple can only be placed (and affect) $A[t+1]$-th block of the sequence, so $A[t+1] := T$.

- **Cash-Register Model**: Another popular model, which allows incremental updates of the sequence blocks as each update at time t has the form of $\langle i, p_i \rangle_t$ where i is the sequence block to update p_i is a positive number (usually, $p_i = 1$) and is *increment-only*. Typically a tuple T at time t contains a multi-set of items that are to be incremented.

- **Turnstile Model**: The turnstile model is the most general data-stream model of them all; it allows updates of multiple sequence blocks at any given time while also allowing negative increments for tuples; in essence though, the main difference between this and the Cash-Register model is that p_i can be either positive, or negative and usually it is $p_i = \pm 1$.

It has to be noted that each problem requires different approaches depending on the streaming model used. Naturally when operating with the turnstile model problems are usually significantly more difficult to solve. This is due to the fact that turnstile model is quite general and allows a lot of flexibility, which in turn presents the designer with significant challenges that need to be resolved.

2.1.2 Data-Stream Approximation Overview

Naturally when we operate in a *streaming* scenario exact results are not feasible due to extraordinary computation and storage requirements, hence methods to perform an approximation of the sequence need to be devised. Depending on the application different approaches to the nature of these techniques are needed and loosely we can group them into the following categories.

2.1.2.1 Approximate Query Estimation

It has been observed that many applications can allow some degree of flexibility in the error bounds of the answer given and that means we can exploit this flexibility to give approximate answers to increase performance This is especially the case for exploratory queries on the data, where the asker would be satisfied with a "close-enough" answer should they of course come in a timely manner. A popular technique that has been extensively used to accelerate the execution times of queries, at the cost of course in

their accuracy, is to answer the query based on its execution into a significantly smaller, carefully selected (or sampled) portion of the dataset. In fact the approximation estimation of queries is the most common use of actual synopsis structures and it is also of particular importance from a computational efficiency and latency standpoint due to the fact that most queries have to be answered in online time. Hence most of the known methods for data-stream approximation have been adapted to solve this problem. Recently Garofalakis et al. [5] used sketches to provide approximate query answers on distributed sliding-window data-streams. Chakrabarti et al. [6] have used multidimensional wavelets in order to provide approximate query answers entirely in the coefficient domain with very promising results. Ioannidis et al. [7] have used histograms to provide approximations of set-valued query answers. Gibbons et al. [8] have used sampling techniques in order to generate stream summary statistics to provide improved answers when executing approximate queries on data-streams. Finally it has to be noted that although approximate error bounds are normally given very recently Agarwal et al. [9] claims that these bounds are not kept when using real query workloads while also proposing a way to accurately detect when it occurs and a potential solution to amend it.

2.1.2.2 Approximate Join-Size Estimation

Another problem that is actively researched in data-streams is the join-size estimation which is usually a particularly challenging problem in streams, especially when the domain of the join attributes is extremely large. A lot of techniques have been introduced to solve this problem, and a vast majority of them use sketching. This is due to the fact that sketching can "*capture*" with enough accuracy the skew of data in various ways, which is particularly important when estimating the join-size of two (or more) attributes. Alon et al. in [10] proposed a solution to track join and self-join sizes when having storage constraints; they employ sketching techniques to create their synopsis data structures while also supporting insertions and deletions with very promising results. Dobra et al. in [11] use again sketching but in conjunction with already gathered statistical information on existing data to improve the quality of the calculated approximations. Again Dobra et al. in [12] use sketch sharing in order to provide better join-size approximations for multiple concurrent queries over data-streams. Ganguly et al. in [13] used skimmed sketches in order to provide approximate answers on data-stream join-aggregates.

2.1.2.3 Aggregation Estimation

For many data-streams one might desire the computation of aggregation metrics such as counts, quantiles and heavy hitters just to name a few. In each case a variety of synopsis

data-structures might be applicable these data structures include but are not limited to sketches, wavelets and others. For example Datar et al. in [14] used sketching alongside exponential histograms in order to provide aggregations such as *average*, *summation*, *histograms* and others over data-streams using the sliding window model. Charicar et al in [15] introduce the *count-sketch* data-structure that allows the discovery of top-k frequent items in a data-stream using very little space and time. Cormode et al. in [16] devised a very clever data structure using sketching called *count-min* which was able to summarize data-streams in *sublinear* space and allows the execution of range, point and inner product based queries while providing answers very fast.

2.1.2.4 Data Mining Applications

These applications, including our own do not require the use of individual data points but instead require a temporal synopsis of the stream which provides a general overview of the data-stream behavior through time. Methods such as clustering [17] and sketches [18] can be used for effective change detection in such use cases. Additionally classification [19] methods can also be used on a supervised synopsis of the data stream. Linear decomposition techniques such as Principal Component Analysis and Singular Value Decomposition have been successfully applied to provide an accurate synopsis of multiple data-streams; indicative works are [2, 20, 21]. Blind Source Separation is closely related with `PCA` and specifically incremental `PCA` in order to track the subspace of the orthonormal basis and has been successfully applied in the past for separating the main trends of the data-streams using methods similar to our own where Papadimitriou et al. presented `SPIRIT` in [22] provide a `PAST` based streaming algorithm for extracting the top-k most dominant patterns out of n streams in $\mathrm{O}(kn)$ space and time. Later Sun et al. in [4] expanded `SPIRIT` while keeping space and time complexity the same into a distributed algorithm `D-SPIRIT` thus scaling the algorithm to monitor a larger amount of streams concurrently. Another very interesting approach is the one of Zhu et al. which presented `StatStream` in [23] where they decomposed thousands of data-streams using `DFT` and by placing their coefficients into a grid structure are able to extract stream correlations very fast and with high accuracy.

2.1.3 Synopsis Design considerations

The designer of each synopsis structure must tailor it to the application domain that needs to be solved. Therefore each synopsis incarnation must be implemented in such a way that is friendly to the needs and constraints that apply to each particular domain that the designer wishes to apply that specific method. For instance a synopsis structure

that will be used for query estimation is much more likely to be very different from a synopsis structure that will be used for data-mining problem such as data-stream drift and trend detection. In general though, we should assume that the desire is to construct a synopsis structure so that it has as wide availability as possible across all the broad classes of problems. This might not always be the case as one might imagine, due to various domain constraints that each problem might have; thus preventing us from using that synopsis for applications in different domains. Loosely the design parameters that one might usually consider when designing a synopsis data-structure are outlined below.

2.1.3.1 Broad Domain Application

The domain of each application is essentially what will have the most significant impact during the synopsis algorithm design and the trade-offs that need to be made. Additionally as it was said previously generic synopsis structures that have broad availability with few adjustments are highly desirable as this can reduce the time and effort one needs to make in order to tailor each structure for a particular problem domain. Of course as one might imagine, if our synopsis algorithm has a narrow field of applications then a different structure would likely be designed for each application. This can increase the time and space requirements for many use-cases, especially in data-mining. This will happen due to the fact that usually we need to extract multiple metrics from the data-streams often requiring more than one synopsis structure to accurately represent all the required attributes, hence having a generic structure that offers us the flexibility required is a very good (and hard to find) trait.

2.1.3.2 Pass Constraints

Due to the high possibility of that our data-set will involve processing multiple, high-dimensional data-streams, more often than not we do not have the luxury of storing some or any data-points for multiple pass processing due to obvious storage and computational limitations. This imposes the restriction that our synopsis algorithm must be able to process our data in a limited number of passes but it is usually only one. This constraint is then transformed to its more strict version called *single-pass* constraint.

2.1.3.3 Time Efficiency

Our synopsis structure must have feasible computational time-complexity, ideally sub-linear, especially when trying to scale the solution both vertically (e.g. number of data-streams processed) and horizontally (e.g. time observed). For example traditional synopsis methods such as histograms have super-linear requirements for their calculation which, of course is not feasible when considering a streaming scenario.

2.1.3.4 Space Efficiency

The case of space requirements for our synopsis must follow a similar bound as with the time-requirements mentioned previously and optimally we would like to have a sub-linear storage requirement with respect to the size of the data-stream(s). Such is the case when using methods such as sketches, where the space complexity is designed to be on the logarithmic scale in the domain-size of our stream(s).

2.1.3.5 Robustness

The synopsis must be accurate enough and ideally have user specifiable error quality bounds ϵ that allow the user to tailor the structure to its needs making the necessary trade-offs that are specific to its problem domain (for example sacrificing synopsis reconstruction accuracy for reduction in processing time).

2.1.3.6 Drift-Awareness

Data-streams are in their nature constantly evolving and it would be unreasonable to assume that changes or *drifts* are unlikely to occur. Hence our algorithms must be able to capture those drifts in order to maintain the quality of our synopsis. Techniques that have been shown to have good results to account for time-drifts in data-streams are exponential forgetting [22, 24], clustering [17] and various other techniques. Carefully designed synopsis structures that handle time-drifts well can and have been be used to perform query or value forecasting [25].

2.1.4 Data-Stream Approximation Techniques

In this section we will briefly summarize the most widely used techniques that are used for data-stream approximation. Most methods stem from other fields such as Signal or Image processing among others and have been adapted to be applicable in data-streams.

2.1.4.1 Sampling

Sampling techniques are among the most basic and simple ways to construct a data-stream synopsis; these synopses are also relatively easy to use with applications into a wide variety of domains due to their not specialized representation and because they use the same multi-dimensional representation as the their originating data streams. In particular most sampling techniques in some way originated from *Reservoir-Sampling* from Vitter et al. [26] and have evolved into techniques such as *Concise-Sampling* by Gibbons et al. [8] and more recently into *Weighted Random-Sampling* by Efraimidis et al. [27].

2.1.4.2 Histogram Generation

Histogram based methods are widely used in many scientific fields and data-streams are not an exception as they provide useful information for the data-stream value distribution. Unfortunately traditional techniques for constructing histograms have super-linear space and time requirements for their calculation and this stems from the use of dynamic programming techniques for optimal histogram generation. Recent adaptations have tried to tackle this obstacle using clever techniques and some indicative work is the calculation in linear time and polylogarithmic space of the Optimal-V histogram with a bounded approximation error $1 + \epsilon$, which was introduced by Guha et al. [28]. Another indicative work is that of Datar et al. [14] where they introduce the notion of exponential-histogram that can be used to maintain accurate stream statistics such as *average*, *summation* over the sliding window data-stream modem with a bounded approximation error ϵ.

2.1.4.3 Sketching

Sketching was introduced in the seminal paper of Alon et al. [10], it has its roots in Wavelet techniques and in fact they should be considered a randomized version of such. Sketches are among (if not) the most space-efficient approximation structure. However, due to the obstacles that present when trying to interpret a sketch based approximation they are not always easy to adapt in an arbitrary application. In fact, the generalization of sketch based methods in the multi-dimensional case is still an open problem. Indicative works regarding sketching are: the *count-min* sketch a novel sketch-based data structure by Cormode et al. in [16], Manku et al. used sketching in [29] to approximate the frequency counts over data-streams and Karl et al. in [30] used sketching to provide a simpler algorithm for finding the most frequent algorithms in data-streams. Another

interesting application of sketches are in probabilistic data-streams and indicative works by Garofalakis et al. [31] and Cormode et al. in [32] present interesting techniques on how to adapt sketches for such use-cases.

2.1.4.4 Wavelet Transformations

Wavelet based transformations have been traditionally used in image and signal processing, namely in JPEG2000 standard [33] and Dirac video codec [34] to name a few notable use-cases. They have also been used in databases for performing a hierarchical decomposition and summarization of its contents. In the streaming scenario wavelets have been adapted mostly using the Haar Wavelet technique; this is due to the fact that Haar Wavelets are easy to implement and has been widely used for successfully performing hierarchical decompositions. The basic idea of wavelet transformations is to perform and maintain a decomposition of the data characteristics into a set of *wavelet* and *basis* functions. The property of such methods is that the higher the order of coefficients the more indicative are for the broad data-trends whereas the more localized and outlier trends are captured by the lower order coefficients. Again indicative works regarding data-streams and wavelets are by Gilbert et al. in [35] where they use wavelet-based approximations while using little space and few computational resources, another interesting work is that of Cormode et al. in [36] where they present a fast approximation of wavelet tracking over data-streams.

2.1.4.5 Linear Transformations

These transformations have been used to summarize the data, by usually performing orthogonal operations to reduce the dimensionality of our data-streams to a more manageable dimension. For example eigenvector decomposition allows us to factorize a matrix into its eigenvalues and eigenvectors; this is extremely useful as we can sort these values in increasing (or decreasing) weight hence extracting the top-k most significant components, additionally if we so desire we can sacrifice accuracy and drop some eigenvalues (and their corresponding eigenvectors) in order to perform a dimensionality reduction. Techniques such as eigenvector decomposition, orthonormal bases, subspaces [3], singular value decomposition [37] are very frequently used and will be discussed later as they are closely related to our method.

2.2 Data Mining on Streams

Data-Mining on streams has attracted a lot of attention from the scientific community in the last decade or so and for obvious reasons; the need for more efficient and scalable methods to extract features from streaming or static data-sets has sparked the interest of researchers and a number of algorithms have already been proposed. In section will describe the related work that has been recently done regarding data-mining on data-streams with particular interest in the scalability and accuracy of each method.

2.2.1 Clustering

This technique has been used with great results in the past; `CluStream` [17] is a great example of data stream clustering due to the fact that it introduced the notion of *micro/macro*-clusters and with the use of both online/offline components achieved great performance and accuracy but due to the heuristic nature of the used algorithm no theoretical quality guarantees are provided. After `Clustream` Aggarwal along with his co-authors developed in [38] a competing system call `HPStream` that bested `CluStream` in almost every benchmark. Recently the density of the data-streams has been studied as a clustering method and indicative works are the DenStream [39] algorithm which combines micro-clustering with a density-estimation process for effective clustering. Another very popular algorithm called k-median has been adapted to be used in data-streams very effectively. An indicative work is that of Charikar et al. in [40] where they propose a solution to overcome the increasing approximation factors that the previous methods had, by increasing the number of levels used which resulted in a divide & conquer algorithm.

2.2.2 Classification

Classification in data-stream mining is incredibly useful as it allows to extract categories from multiple data-streams. For example ideal use-case for classification would be real-time decision support in business analytics, critical astronomical applications or monitoring of cell phone activity in a city for security reasons are important and ideal use cases for classification. Additionally classifiers can be tailored into taking in account the infinite length and evolving nature of streams as it was shown by Wang et al in [41] who developed a scalable system for classifying drifting data-streams. They perform this using a novel technique using weighted classifier ensembles for mining the streams. Aggarwal et al. in [19] introduced a novel on demand classification algorithm for data-streams, which adapt the idea of *micro-clustering* from `CluStream` into the classification

domain, with very promising results. Another notable mention is that of Domingos et al. in [42] where they introduce `VFDT` or *Very-Fast-Decision-Tree*; which is a tree-based learning system based on Hoeffding trees, again with promising results.

2.2.3 Frequency Counting

Frequency counting is a very interesting topic for due to the fact that can display with accuracy the skew of data and in many applications this is a desirable feature to extract. Cormode et al. in [43] present a novel technique that uses randomized algorithms to track the "*hot*" (i.e. most frequent) items and is designed to be used in the turn-stile data-stream model, which computationally is the hardest to analyze. Another interesting work is that of Manku et al. in [29] where he devises a novel algorithm for approximate frequency counts that uses all the previously observed historical data to calculate the patterns that are frequent in the monitored data-streams incrementally. Giannela et al. in [44] have devised an algorithm that discovers the most frequent item-sets over a data-stream. They use incrementally maintained *time-tilted* windows for each frequent pattern at multiple time granularities in conjunction with an *FP-Tree* inspired data-structure to store each frequent pattern found; it has to be noted that this method does not use any of the most commonly synopsis methods of today.

2.2.4 Time Series Analysis

Time-Series analysis has been used due to the fact that its definition is almost identical to the definition of a data-stream, hence most of the study models used in this field are, usually with modifications, applicable to data-streams as well. Indyk et al. in [45] have devised and proposed an algorithm that uses sketches in *pools* to produce approximate solutions with probabilistic error bounding to two problems in the time-series domain, the relaxed periods and average trends. Zhu et al. in [23] have devised a clever decomposition of data-streams using `DFT` that they can extract fast and accurate the correlations between data-streams. Additionally Cole et al. in [46] proposed a sketching method in order to find the stream correlations with greater accuracy and space complexity when we are dealing with uncooperative time-series. Chen et al. in [47] introduced the notion of *regression-cubes* for data-streams; they have proposed to use multidimensional regression analysis in order to create a compact cube that could be used for answering aggregate queries over the monitored data-streams. Papadimitriou et al. in [22] have used Yang's `PAST` algorithm to extract the top-k most dominant patterns; they define a pattern as the projection of each eigencomponent onto the current monitored-stream

values. Sakurai et al. in [48] have introduced `BRAID`, an elegant framework to find the *group-lag* correlations amongst many data-streams.

Chapter 3

Problem Statement

In this chapter we will give a formal definition of the problem that we are trying to solve, while also explaining the techniques used in order to create our algorithm. After formalizing both the problem and techniques used we will present our algorithm that provides an adequate solution to the problem.

3.1 Preliminaries

In this problem we model each data-stream using the *time-series* model and as such a stream data-source, for our purposes is an unbounded sequence of tuples in time-order. For specificity we assume that such data-streams produce one tuple per time-unit. To find such patterns more efficiently a sound technique is to find a method to distribute the workload amongst many workers. This is performed by splitting the input data-streams into *groups* and incrementally finding the patterns in each of the groups. As one might think, finding the patterns in each of the groups is considerably less expensive than finding all the patterns at once; this step is called *local-pattern-discovery*. Then by having discovered each of the groups' *local-patterns* find efficiently the *global-patterns* that are representative for all of our monitored streams.

For the purpose of this thesis we view the original data-streams as points in a high-dimensional space, where as stated above produce one tuple per time-unit. After grouping the streams; each of the groups' local-patterns are extracted using low-dimensional projections of the original points. It has to be noted that for reconstruction purposes the basis of the low-dimensional spaces is kept and incrementally updated for each group. Another noteworthy fact is that the tuples of each group are assumed to be synchronized

in the time-axis; if a value is not received within a specified window it is assumed to be lost while also possibly indicating malfunction[1].

3.2 Problem Formalization

Given m groups of data-streams that each one consists of $\{n_1, ..., n_m\}$ co-evolving numeric data-streams, we want to devise an elegant solution that will solve the following two problems:

- Incrementally find the top-k most dominant patterns $L_i \in \mathbb{R}^{1\times k}$ within a single group $i \in [1, m]$ using F_L function.
- Efficiently combine the each one of the m local patterns L_i to find the representative global patterns $G \in \mathbb{R}^{1\times k}$ using F_G function of all monitored streams in each group.

Before we introduce these two aforementioned functions we have to first describe the data-stream distribution amongst each group. Let a group be denoted as S then each i-th group S_i is comprised out of an unbounded sequence of n_i-dimensional vectors where n_i is the number of data-streams contained in the group S_i with $1 \leq i \leq m$ and m being the total number of monitored groups. S_i can also be represented as a matrix with n_i columns and an unbounded number of rows. The *intersection* of matrix S_i, defined as $S_i(t, l)$ is the t-th row and the l-th column of S_i; this represents the value of the l-th stream recorded at time t in the i-th group. Finally using the definitions described above we are now able to define the functions for monitoring per-group *local-patterns* F_L and the aggregated *global-patterns* F_G.

Function F_L definition for *local-pattern* calculation:

$$F_L : (S_i(t + 1, :), B) \rightarrow L_i(t + 1, :) \tag{3.1}$$

F_L takes as input $S_i(t+1, :) \in \mathbb{R}^{1\times n_i}$ vector which contains current values for each data-stream monitored by the group i at time $(t+1)$ and the block-size that we current have, usually this is a static value throughout the execution of our system.

The function F_L has to incrementally maintain the *local-patterns* $L_i \in \mathbb{R}^{1\times d}$ which is then shared with the aggregators in order to produce the *global-patterns* in a way that will be explained later. In a similar fashion like F_L the definition for the *global-pattern* detection function F_G follows in full.

[1] usually it is not a fault of the actual system but the stream producer (e.g. *spout*)

$$F_G : (L_1(t+1,:), L_m(t+1,:)) \rightarrow G(t+1,:) \tag{3.2}$$

The F_G function takes as arguments all the *local-patterns* $L_i, i \in [1, m]$ for each one of the m groups that were generated at time $(t+1)$ and produces a *global-pattern* vector $G(t+1,:)$ at each time-step which is then propagated to each one of the m groups and is also our final output that holds the top-k most dominant trends of all monitored streams.

3.3 PCA Decomposition

Principal component analysis is a linear transformation, specifically it uses an orthogonal transformation to covert a set of observed possibly correlated values into a set of values of linearly uncorrelated variables that are called *principal components*. The number of principal components of a *lossless* transformation is the same as the number of the original variables before performing `PCA`. The dimensionality reduction comes when we are allowed to perform a *lossy* `PCA` transformation, this can be done easily as the principal components can be sorted based on their significance using the eigenvalues; this allows to "drop" a number of non-significant principal components in order to perform the dimensionality reduction. Hence `PCA` transformation is a great way to reduce the dimensionality of data and would ideally suit our needs; yet the classic method for computing `PCA` is limited by the prohibitive cost of forming the empirical covariance matrix, which typically is when having a stream of n-dimensional vectors a $n \times n$ dense matrix that in turn requires $\mathrm{O}\left(n^2\right)$ space. The quadratic cost of space is, as one might imagine, barred for large datasets or in *streaming* scenarios and poses a major bottleneck for its potential applications.

The output of `PCA` when operating on streaming vectors of n-dimensions is a set of $k \in [1, n]$, n-dimensional principal components which span the subspace created by `PCA` transformation. Naturally a *lossless* application of `PCA` means that $k = n$ and would result in a $n \times n$ matrix but a relaxation of the *lossless* decomposition is expected and we would be allowed to have a significantly reduced number of principal components. Mitliagkas et al. in [2] proposed a novel algorithm with strict theoretical bounds on approximation quality that required $\mathrm{O}\left(kn\right)$ space which by definition is the lowest possible when performing `PCA`. They base their method in the classic Power-Method as described in [49] and adapt into a block-wise stochastic variant with great results. The intuition behind their algorithm stemmed from the fact that most stochastic methods for `PCA` approximation had a variable and possible large variance at each step and hence

the standard concentration of inequalities would give vacuous bounds. Their proposed method used a block-wise algorithm that had a variance reduction step built in; in essence they would update the basis Q_τ of `PCA` once every block using a QR-decomposition while within each block they would average-out the noise thus reducing the variance contained in processed values. The proposed algorithm they devised is shown in Algorithm 1 which follows in full below.

Algorithm 1 PCA using the Block-Stochastic Power Method

Input: $\mathcal{X} = x_1, ..., x_n$ Block size: B

1: $H^i \sim \mathcal{N}(0, I_{p\times p})$, $i \leq i \leq k$ ▷ Initialization
2: $H = Q_0 R_0$ ▷ Get initial Q_τ basis by QR-Decomposition
3: **for** $\tau = 0, ..., n/B - 1$ **do**
4: $S_{\tau+1} \leftarrow 0$
5: **for** $t = B\tau + 1, ..., B(\tau + 1)$ **do**
6: $S_{\tau+1} \leftarrow S_{\tau+1} + \frac{1}{B} x_t x_t^T Q_\tau$ ▷ Within block, average out noise.
7: **end for**
8: $S_{\tau+1} = Q_{\tau+1} R_{\tau+1}$ ▷ Update basis Q_τ using QR-Decomposition
9: **end for**
output

The subspace basis approximation at each step is contained in Q_τ and as we see is updated once per block when performing the *QR-decomposition* at line 8. Due to its performance and high accuracy we selected this algorithm to perform the `PCA` approximation in each one of the m nodes.

3.4 Subspace Tracking

Subspace tracking is was introduced by Yang in [3] and Sun et al. in [4] adapted it to be used for tracking these projections in a distributed manner. Each pattern is the defined as the projection of each column of the participation matrix $W_t \in \mathbb{R}^{k\times n}$ against the remainder of $x \in \mathbb{R}^{1\times n}$ and in total we would have k numbers each being the representative projection value for each principal component. This method is closely related to `PCA` but again, unlike the classic method of performing `PCA` it does not require the entire data-set in order to find the low-rank approximations as well as the participation weights correctly; hence this method is incremental only requiring the current values of the data-streams at time t. Also the algorithm is resilient enough to be able to accurately converge to the correct participation weights even when data-streams of $S_i(t, :)$ do not

follow any particular distribution. The incremental update algorithm for *local-pattern* discovery that is used in `D-SPIRIT` is the following:

Algorithm 2 Local Pattern Discovery in `D-SPIRIT`

Input: new vector of values $S_i(t+1,:)$, old global patterns $G(t,:)$
Output: local patterns $L_i(t+1,:)$

$\overrightarrow{x_1} := S_i(t+1,:)$ ▷ Initialize to new values
for all $1 \leq j \leq k$ **do** ▷ At each time-step do for all k patterns
  $y_j := \overrightarrow{x_j} W_{i,t}(j,:)^T$ ▷ project $\overrightarrow{x_j}$ onto $W_{i,t}(j,:)$
  if $G(t,:)$ is *null* **then** ▷ boundary case, use y_j instead as $G(t,:)$
    $G(t,:) := y_j$
  end if
  $d_j \leftarrow \lambda d_j + y_j^2$ ▷ local energy E, determines the update magnitude
  $\overrightarrow{e} := \overrightarrow{x_j} - G(t,j) W_{i,t}(j,:)$ ▷ the error, $\overrightarrow{e} \perp W_{i,t(j,:)}$
  $W_{i,t+1}(j,:) \leftarrow W_{i,t}(j,:) + \frac{1}{d_j} G(t,j) \overrightarrow{e}$ ▷ update participation weight
  $\overrightarrow{x_{j+1}} := \overrightarrow{x_j} - G(t,j) W_{i,t+1}(j,:)$ ▷ repeat with the rest of $\overrightarrow{x}$
end for
return $L_i(t+1,:) := S_i(t+1,:) W_{i,t+1}^T$ ▷ The updated i-th patterns

For each j (out of k patterns) $\overrightarrow{x_j}$ is the component of $S_i(t+1,:)$ in the orthogonal complement of the space that is spanned by the updated weight $W_{i,t+1}(j',:)$, $1 \leq j' \leq j$. The column vectors of $W_{i,t+1}$ are ordered based on their contribution significance to the patterns; more specifically they are actually ordered based on their decreasing eigenvalue magnitude (energy). It also is shown in both that under stationary assumptions we have a very quick convergence to the actual principal directions.

3.5 Local Pattern Discovery

Taking ideas from both Mitliagkas et al. [2] from their block-based `PCA` algorithm as well as the projection tracking from Yang [3] in order to create our own algorithm, we extended these two concepts to use the higher quality `PCA` approximation while discovering the top-k most dominant patterns by projecting the values of S_i onto the current basis approximation Q_τ.

3.5.1 Local Pattern Disocvery for fixed k

As it was previously stated the function F_L within each group is responsible for the calculation at each time-step for the low-dimensional projections $L_i(t,:)$ and to update

the subspace basis approximation Q_τ if it is needed as it only happens once per each block. Generally we need a metric to measure the quality of the stream reconstruction at any given time, this is accomplished by ensuring that the square-difference which is defined as follows: $||S_i(t,:) - \hat{S}_i(t,:)||^2$ is predictably small. The reconstruction of $S_i(t,:)$ is defined as $\hat{S}_i(t,:) = L_i(t,:) \times Q_{i,\tau}^T$. The initial algorithm for detection of the top-k most dominant patterns using a fixed k follows in full.

Algorithm 3 Local Pattern Discovery in `NeuralStream` (fixed k)

Input: $\mathcal{S} = s_1, ..., s_n \in \mathbb{R}^{p \times 1}$ Block size: B

1: $H^i \sim \mathcal{N}(0, I_{p \times p})$, $i \leq i \leq k$ ▷ Initialization
2: $H = Q_0 R_0$ ▷ Get initial Q_τ basis by QR-Decomposition
3: **for** $\tau = 0, ..., n/B - 1$ **do**
4: $D_{\tau+1} \leftarrow 0$
5: **for** $t = B\tau + 1, ...B(\tau + 1)$ **do**
6: $D_{\tau+1} \leftarrow D_{\tau+1} + \frac{1}{B} s_t s_t^T Q_\tau$ ▷ Within block, average out noise.
7: $L_t = -s_t^T Q_\tau$ ▷ calculate the projections
8: **end for**
9: $D_{\tau+1} = Q_{\tau+1} R_{\tau+1}$ ▷ Update basis Q_τ using QR-Decomposition
10: **end for**
output

3.5.2 Local Pattern Discovery for variable k

Algorithm 3 that is shown above has a *fixed* number of k-dimensions and in turn principal directions. Due to the fixed number of k we cannot guarantee that the reconstruction error will be predictably small over time at all times as we have no way of adjusting the number of principal directions *on-the-fly* depending on the reconstruction quality. Hence the number of *local-patterns* per group needs to be adjusted over time (either increased or decreased) as we cannot possibly expect to know beforehand the number of patterns to expect.

As indicated in [50] a suitable method to obtain the number of needed patterns on the fly is *Energy-Thresholding*; thus we employ a technique that is based on that principle as it helps us to estimate the number of principal components needed to accurately represent the monitored streams at any given time. In practice we need to track *two* quantities in order to successfully apply this method; the first one is the actual total captured energy of the streams per group, formally given by Equation 3.3 that follows.

$$E_{i,t} := \frac{1}{t}\sum_{\tau=1}^{t} ||S_i(\tau,:)||^2 = \frac{1}{t}\sum_{\tau=1}^{t}\sum_{j=1}^{n_i} S_i(\tau,j)^2 \tag{3.3}$$

The second quantity that we need to keep track of is the total energy $\hat{E}_{i,t}$ captured by the reconstruction approximation $\hat{S}_i(t,:)$; which in a similar fashion as in Equation 3.3 above is given by Equation 3.4 that follows.

$$\hat{E}_{i,t} := \frac{1}{t}\sum_{\tau=1}^{t} ||L_i(\tau,:)||^2 \tag{3.4}$$

Lemma 1. *Assuming that the basis at time* τ, $Q_\tau \in \mathbb{R}^{p\times k}$ *has orthonormal columns, and the patterns of the* i*-th group again at time* τ *are* $L_i(\tau,:) \in \mathbb{R}^{1\times k}$ *then we have*

$$\hat{E}_{i,t} := \frac{1}{t}\sum_{\tau=1}^{t} ||\hat{S}_i(\tau,:)||^2 = \frac{1}{t}\sum_{\tau=1}^{t} ||L_i(\tau,:)\times Q_{i,\tau}^T||^2 = \frac{1}{t}\sum_{\tau=1}^{t} ||L_i(\tau,:)||^2 \tag{3.5}$$

Proof. Since the columns of basis transpose Q_τ^T are also orthonormal, then we can easily show that the Euclidean norm of the reconstruction $\hat{S}_\tau$ is $||\hat{S}_\tau||^2 = ||L_{\tau,1}Q_{\tau,1}^T + \cdots + L_{\tau,k}Q_{\tau,k}^T||^2 = L_{\tau,1}^2||Q_{\tau,1}^T||^2 + \cdots + L_{\tau,k}^2||Q_{\tau,k}^T||^2 = L_{\tau,1}^2 + \cdots + L_{\tau,k}^2 = ||L_\tau||^2$. The final results follows by summing over τ. □

The above formula concludes to that result assuming $Q_{i,\tau}$ has orthonormal columns; although we can further simplify it to this form:

$$\hat{E}_{i,t} := \frac{1}{t}\sum_{\tau=1}^{t} ||L_i(\tau,:)||^2 = \frac{t-1}{t}\hat{E}_{i,t-1} + \frac{1}{t}||L_i(t,:)||^2 \tag{3.6}$$

This form is much more convenient and shows the incremental nature of updating the energy captured; it is this form that is used in our implementation. To accurately adhere to each of the monitored groups' pattern features we use a separate threshold for each of the groups; to that end we use a low-energy and a high-energy threshold called f_E and F_E respectively. The actual retained energy range is always between: $[f_E \cdot E_{i,t}, F_E \cdot E_{i,t}]$ and whenever the boundaries are exceeded we increase or decrease the number of patterns kept accordingly. Now let's prove that the actual captured energy by the reconstruction $\hat{S}$ is indeed bound within the specified interval $[fE, F_E]$.

Lemma 2. *The relative squared error of the reconstruction* $\hat{S}$ is bounded by the following inequality:

$$1 - F_E \leq \frac{\sum_{\tau=1}^{t} ||\hat{S}_\tau - S_\tau||^2}{\sum_t ||S_\tau||^2} \leq 1 - f_E \tag{3.7}$$

Proof. From the orthogonality of S_τ and the complement $\hat{S_\tau} - S_\tau$ we have that the quantity: $||\hat{S_\tau} - S_\tau||^2$ is equal to $||S_\tau||^2 - ||\hat{S_\tau}||^2$ which by using *Lemma 1* can be transformed as: $||S_\tau||^2 - ||L_\tau||^2$. Then the results follows from the definitions of E, $\hat{E}$ and by summing over τ. □

Based on what was previously mentioned the updated version of our algorithm using the captured stream energy as a threshold measurement in order to increase/decrease the number of tracked principal components dynamically is presented in full below.

Algorithm 4 Local Pattern Discovery in `NeuralStream` (variable k)

Input: $\mathcal{S} = s_1, ..., s_n \in \mathbb{R}^{p \times 1}$ Block size: B

1: $H^i \sim \mathcal{N}(0, I_{p \times p})$, $i \leq i \leq k$ ▷ Initialization
2: $H = Q_0 R_0$ ▷ Get initial Q_τ basis by QR-Decomposition
3: **for** $\tau = 0, ..., n/B - 1$ **do**
4: $D_{\tau+1} \leftarrow 0$
5: **for** $t = B\tau + 1, ...B(\tau + 1)$ **do**
6: $D_{\tau+1} \leftarrow D_{\tau+1} + \frac{1}{B} s_t s_t^T Q_\tau$ ▷ Within block, average out noise.
7: $L_t = -s_t^T Q_\tau$ ▷ calculate the projections
8: $E_t \leftarrow \frac{\lambda(t-1)E_t + s_t^2}{t}$ ▷ Update actual energy
9: $\hat{E}_t \leftarrow \frac{\lambda(t-1)\hat{E}_t + L_t^2}{t}$ ▷ Update approximation energy
10: **end for**
11: **if** $\hat{E}_t < f_E E_t$ **then** ▷ Check if we need to increase k
12: Initialize a new column for S_τ and left-pad it, resize all tables
13: **else if** $\hat{E}_t > F_E E_t$ **then** ▷ Check if we need to decrease k
14: Drop the right-most column of all tables.
15: **end if**
16: $D_{\tau+1} = Q_{\tau+1} R_{\tau+1}$ ▷ Update basis Q_τ using QR-Decomposition
17: **end for**
output

As we can see at each time step we calculate the representative *local-patterns* L_{t+1} and when needed we resize our tables accordingly so that we increase or decrease the monitored patterns based on the actual energy that our reconstruction is able to capture. It has to be noted that both for performance and noise-reduction reasons the adjusting step is only performed at the end of each block.

3.5.3 Making Local Pattern Discovery more Robust

Another issue that arises is, that when operating in the *streaming* case time value can get extremely high, extremely fast; this rapid increase will create a lot of number stability problems in the long run concerning the captured energy of the streams. Thankfully by altering a bit the equations and using the reconstruction/actual energy *ratio* helps us alleviate these problems. Thus instead of keeping track the normalized energies at time t we keep the actual aggregated value of the energies at time t.

The actual total captured energy at time t is given by:

$$E_{total} := \sum_{\tau=1}^{t} ||S_i(\tau, :)||^2 \tag{3.8}$$

The reconstruction total captured energy at time t is given by:

$$\hat{E}_{total} := \sum_{\tau=1}^{t} ||L_i(\tau, :)||^2 \tag{3.9}$$

Now to actually avoid this problem we have to use exponential forgetting $\lambda \in (0, 1)$ factor in order to "forget" the energy captured so that we don't run to any number stability problems while also keeping the quality of the reconstruction within the desired bounds. To achieve that we exploit the fact that the following holds true.

Lemma 3. *Given a sequence of the form:* $x_i = \lambda x_i + y$ with $x_0 = 0$, $\lambda \in (0, 1)$ and y a scalar value then the value of x_i asymptotically will have the value $\frac{y}{\lambda - 1}$. The rate that this convergence happens is determined by the value of λ factor.

Proof. Expanding the recursive relation above we can easily find that asymptotically it has the following solution: $x_i = \frac{y(\lambda^i - 1)}{\lambda - 1}$. Since $i \to \infty$, $\lambda^i \to 0$ hence $x_i = \frac{y}{1-\lambda}$. □

Using *Lemma 3* we transform the relations so that x_i is asymptotically as close as possible to the value of y; typical values in that case would be in the range of $\lambda \in [0.9, 1)$.

The total captured energy by the actual observed values at time t using exponential forgetting with a λ factor is given by:

$$E_{total} = \lambda \underbrace{\sum_{\tau=1}^{t-1} ||S_i(\tau, :)||^2}_{E_{total,t-1}} + \underbrace{||S_i(t, :)||^2}_{E_t} \tag{3.10}$$

The total captured energy of the reconstruction at time t using exponential forgetting with a λ factor is given by:

$$\hat{E}_{total} = \lambda \underbrace{\sum_{\tau=1}^{t-1} ||L_i(\tau,:)||^2}_{\hat{E}_{total,t-1}} + \underbrace{||L_i(t,:)||^2}_{\hat{E}_t} \tag{3.11}$$

So instead of checking if the actual normalized energy of the reconstruction at the end of each block B which occurs at time $\tau \in [B, nB]$ is within the bounds specified; we now have to ensure that at the end of each block (in other words at the same time τ as before) that the following inequality holds true:

$$f_E \leq \underbrace{\frac{\hat{E}_{total}}{E_{total}}}_{E_{ratio}} \leq F_E \tag{3.12}$$

Now the updated algorithm for the local-pattern detection function F_L using the notions and techniques outlined above for making the stream energy capture more robust follows in full.

Algorithm 5 Local Pattern Discovery in `NeuralStream` (variable k, better E capture)

Input: $\mathcal{S} = s_1, ..., s_n \in \mathbb{R}^{p \times 1}$ Block size: B

$H^i \sim \mathcal{N}(0, I_{p \times p})$, $i \leq i \leq k$ ▷ Initialization
$H = Q_0 R_0$ ▷ Get initial Q_τ basis by QR-Decomposition
for $\tau = 0, ..., n/B - 1$ **do**
    $D_{\tau+1} \leftarrow 0$
    for $t = B\tau + 1, ...B(\tau + 1)$ **do**
        $D_{\tau+1} \leftarrow D_{\tau+1} + \frac{1}{B} s_t s_t^T Q_\tau$ ▷ Within block, average out noise.
        $L_t = -s_t^T Q_\tau$ ▷ calculate the projections
        $E_{total,t} \leftarrow \lambda E_{total,t} + s_t^2$ ▷ Update actual energy
        $\hat{E}_{total,t} \leftarrow \lambda \hat{E}_{total,t} + L_t^2$ ▷ Update approximation energy
    end for
    $E_{ratio,t} \leftarrow \hat{E}_{total,t} / E_{total,t}$ ▷ Update the energy ratio at time t
    if $E_{ratio,t} < f_E$ **then** ▷ Check if we need to increase k
        Initialize a new column for S_τ and left-pad it, resize all tables
    else if $F_E < E_{ratio,t}$ **then** ▷ Check if we need to decrease k
        Drop the right-most column of all tables.
    end if
    $D_{\tau+1} = Q_{\tau+1} R_{\tau+1}$ ▷ Update basis Q_τ using QR-Decomposition
end for
output

3.6 Global Pattern Discovery

Now we will describe in detail the method for discovering the aggregated *global-patterns* amongst all monitored groups; in essence, we will now formally give the declaration of function F_G. *Global-patterns*, in a similar fashion as *local-patterns* are the low-dimensional projections of all the monitored streams across all groups. This is essentially having to discover all of the patterns from all the monitored streams centrally but using decentralized computation. The complete definition of function F_G that is responsible for *global-pattern* discovery follows in full.

Algorithm 6 Global Pattern Discovery in `NeuralStream`

Input: all *local-patterns* $L_1(t,:), ..., L_m(t,:)$

Output: current *global-patterns* $G(t,:)$

1: $k \leftarrow \max(k_i)$
2: **for** $1 \leq j \leq k$ **do**
3: $\quad G(t,j) := \sum_{i=1}^{m} L_i(t,j)$
4: **end for**

The above algorithm can be formulated as $G(t,:) = F_G([L_1(t,:), ..., L_m(t,:)])$, this is true due to the fact that the *global-patterns* are essentially the summation of all the *local-patterns* of each group; in other words $G(t,:) = F_G([L_1(t,:), ..., L_m(t,:)])$ is equivalent to the following statement $G(t,:) = \sum_{i=1}^{m}(L_i(t,:)$. Of course for this to work all of the $L_i(t,:)$ must have the same number of principal components. In the case that it's not we can easily alleviate the problem by expanding the $L_i(t,:)$ from the groups that have fewer principal components to match the dimensions of the group having the most (without of course polluting the result).

3.7 Tree Expansion

As a natural expansion to the scheme outlined in [4] described, the algorithm modification that is proposed is the following: Let's assume that we have a n-level hierarchy and that each level of the hierarchy has one or multiple groups with the only restrictions being i) that each level must have less groups from its previous one and ii) each group must only propagate its patterns to one of the available groups from the next level; using this recursive definition we can build a tree that propagates the patterns distributing their discovery amongst many levels enabling our system to scale better and with greater flexibility.

Lemma 4. *Let's assume, without loss of generality, that we have a n-level tree such that at each depth its nodes are data-stream groups as defined previously and that each of the tree levels have fewer groups than their parent level (starting from top to bottom). Then by applying function* F_L *at the tree leafs and* F_G *at each subsequent upper-level we can discover all common patterns across all levels of the n-level tree. and in essence the definition is transformed as follows* $G(t,:) = F_{i,G}(F_{i-1,G})$ *which in turn can be expressed as* $G(t,:) = \sum_{i=1}^{n} \sum_{j=1}^{m_i} G_{i,j}(t,:)$. *(*$m_i$ *is the number of groups each tree level has).*

Proof. The tree will have at least two levels, which is the basic level hierarchy that is presented in [4]; assuming that we have more than two-stages we can decompose easily the tree to multiple 2-stage equivalent hierarchies. This can be done by following the restrictions outlined above with an additional rule: that we can only join (two) consecutive levels to perform the decomposition; for example should we have a 3 stage level tree we can decompose its to the following 2-stage equivalents the first one containing levels: 1 & 2 and second: 2 & 3. The correctness of the 2-stage tree scheme is proved in [4], hence further action is not needed. □

Chapter 4

Implementation

This chapter will first describe the tools we used in order to implement the `NeuralStream` framework. Then each each component implementation that comprises our framework will be detailed and analyzed. Finally we will outline all the implementation-specific design choices that we made along the way in order to increase robustness, speed or efficiency.

4.1 Lambda Architecture

This architecture for data-processing systems that `Storm` was designed to be a part of, well devised in order to be able to handle massive amounts of data by taking advantage of both *batch* and *stream* processing frameworks. This approach is an attempt to balance the constraints and trade-offs that one can make when asking a particular query. For example one might require an answer of high accuracy to the query submitted into the system but that would not be possible using the *stream* processing framework, so instead this query would be answered by the *batch* processing framework with higher latency of course. Instead most answers are able to have relaxed constraints and would be answered by the *streaming* framework. Additionally the *streaming* framework can be enabled to use some of the previously computed answers by the batch processing framework to boost its response quality.

This architecture is comprised out of three distinct components which are the following:

- **Batch Layer**: The batch layer uses a framework that provides higher latency answers and (usually) of higher quality than the streaming one. Hence this layer aims at providing excellent accuracy by being able to afford to process *all* available

data when answering a query. An example framework that would be used in this layer of the architecture would be `Apache Hadoop` [51].

- **Speed Layer**: This layer attempts to process data-streams in real time and without the quality requirements that an answer of the batch layer would have. This layer would ideally have a very high throughput as it aims to minimize the latency of answer response by providing that answer by only processing a recent portion or synopsis of the observed data. Typically answers provided by this layer have very small latency and are not as accurate; our selected framework `Storm` was designed to be an ideal candidate for that role.

- **Servicing Layer**: The output of both layers are stored within the *servicing* layer, which is responsible for handling the ad-hoc queries by returning the precomputed or executed answers that each query requires. Example technologies of that include `Apache Cassandra` [52], `Apache HBase` [53] amongst others.

A higher view of the aforementioned architecture is presented in Figure 4.1 that follows.

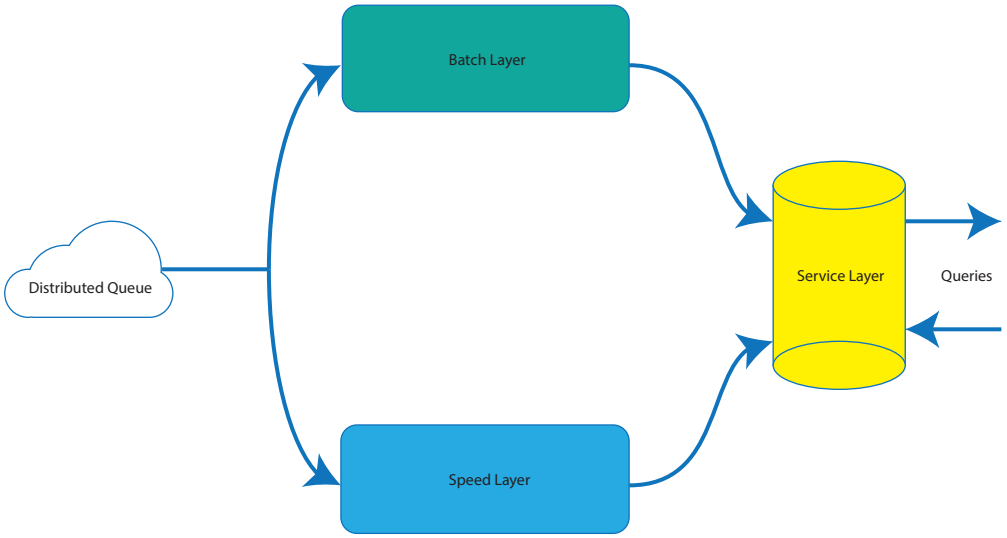


FIGURE 4.1: Lambda Architecture Layout

This application only focuses on the `Speed-Layer` of this architecture and is implemented as such, while the servicing layer at this point is a *web-service* that provides access to the processing results.

4.2 Apache Storm

`Apache Storm` [1] initially developed by Nathan Marz at Twitter and now an official `Apache Foundation` project, is a free *open-source* project that provides an elegant framework for performing fast, scalable yet efficient distributed real-time computations using unbounded data-stream sources. It's a *polyglot*-enabled framework and bindings for a lot of languages already exist. It also allows great flexibility of input data sources, as it can be used with a number of popular distributed, high-throughput reliable queuing systems such as `Kafka`, `Kestrel` and others. The great flexibility, proven robustness and reliability made the selection of `Apache Storm` as the framework on top of which to develop `NeuralStream` a very ideal choice.

4.2.1 Streams in Storm

A stream in `Storm` is defined as the unbounded sequence of tuples that are processed and created in parallel in a distributed fashion; it also is the main data *abstraction* that the framework provides. To define a data stream in `Storm` one must first define the stream's *schema*, which names the fields in each of the stream's tuples. Stream Tuples can contain in each field any data-type that is *serializable*; if one is not (such as many user defined `Java` classes) then one can write a custom serializer to make this functionality possible.

Each stream in a topology is assigned with a unique id when declared; in the very common case of a topology having only *single-stream* `spouts` and `bolts` they are all assigned the default id by `Storm`. An example stream that shows the consecutive tuples as well as an indicative tuple schema is shown in Figure 4.2.

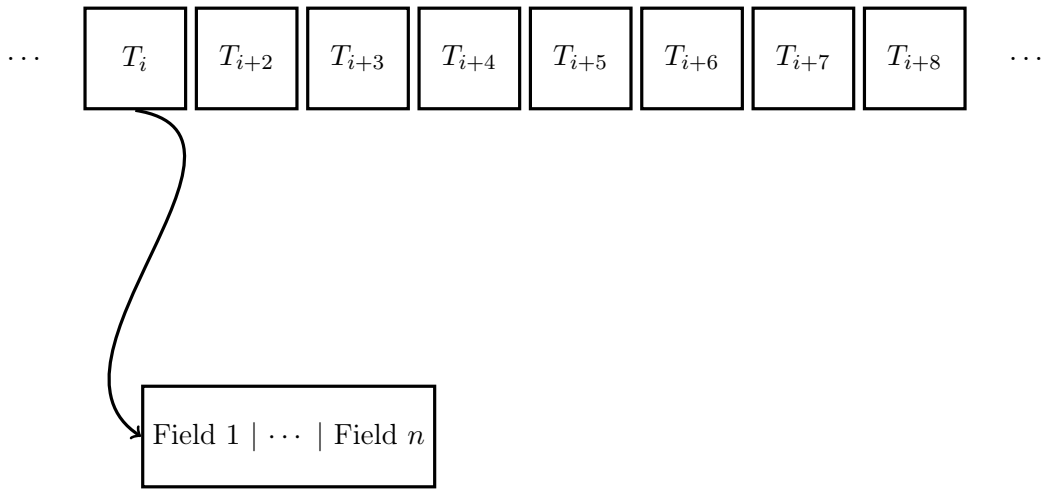


FIGURE 4.2: Example `Storm` Tuple Stream

4.2.2 Storm Architecture Introduction

On a higher level `Storm` actually only exposes to its users the notion of *topologies* along with its two basic primitives. `Storm` has two basic computational units, `spouts` which are *only* content *producers* and `bolts` which can be *consumers* as well as *producers*.

These two basic primitives can be employed to perform tasks such as transformations, accumulations and many other things on the data contained in the streams flowing through the system. Finally these primitives are connected to each other and they form a `topology`, which essentially defines the path of each stream within the system and its layout cannot be changed without first restarting the affected `topology` (but not `Storm` itself). A typical `topology` can be viewed in the following Figure 4.3, where taps represent `spout` primitives while the others represent `bolt` primitives.

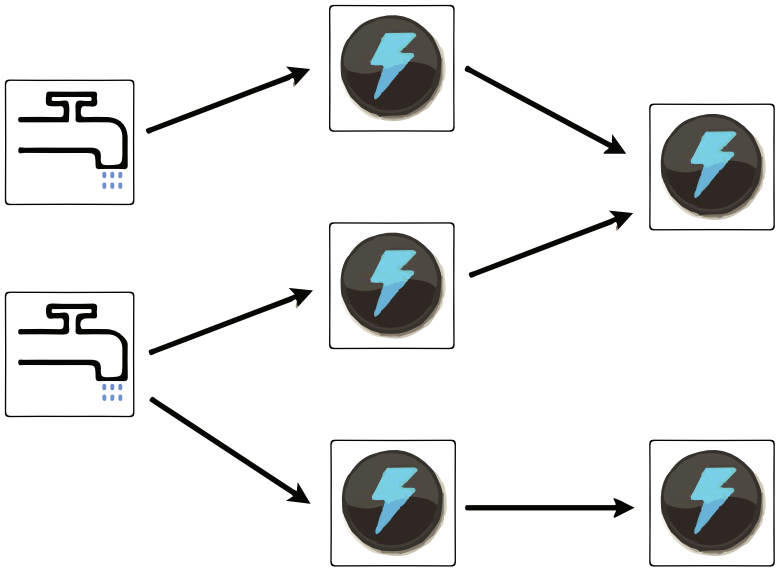


FIGURE 4.3: Typical Storm topology

4.2.3 Storm Architecture Detailed

`Storm` has multiple components in order to provide the advertised functionality. Actually architecturally `Storm` shares a lot of design phylosophies with another very popular framework for big-data processing `Hadoop`; albeit `Hadoop`'s primary focus is *batch-processing* whereas `Storm` is used primarely for *real-time* applications. The main difference between `Hadoop` and `Storm` from a user's perspective is that when a job is assigned in `Hadoop` it has to finish at some point, whereas when a `topology` is started on a `Storm` cluster it will never stop running, unless it is terminated by the user.

There are two kinds of nodes that reside in a `Storm` cluster; the *master*-node and the *worker*-nodes. The *master*-node runs a daemon that's called *Nimbus* which is analogous to `Hadoop`'s *JobTracker* daemon. The *Nimbus* daemon is responsible for distributing the `topologies` code around the cluster nodes, assigning worker tasks and doing failure monitoring. Each of the worker nodes that are in the cluster execute another daemon called the *Supervisor*; this daemon listens for potential *Nimbus* orders, such as what to execute and when. This means that each *Supervisor* daemon is allowed to assign, start, stop as necessary any number of worker processes within each node based on what *Nimbus* has assigned to it. Naturally, usually only a subset of a `topology` is executed in each node; hence running a `topology` actually consists of spawning many worker processes across the cluster nodes so that each executes a portion of the `topology`'s total workload.

Master and *Worker* nodes must be able to communicate, more accurately *Nimbus* must be able to receive and deliver reliably, messages to the *Supervisors* that run on each cluster node and vice-versa. To achieve reliable coordination amongst the *Supervisor* and *Nimbus* daemons `Storm` uses another `Apache Foundation` project, the `Zoopkeeper` [54]; which is quoted from the project's page "a centralized service for maintaining configuration information, naming, providing distributed synchronization as well as providing group services" . It has to be noted that to boost the reliability of `Storm`, both *Nimbus* and *Supervisor* daemons have been designed as *stateless* and *fail-fast* while their state is kept within the `Zookeeper` cluster, stored in a physical disk or both. Figure 4.4 shows this architecture structure and how its interconnected components communicate.

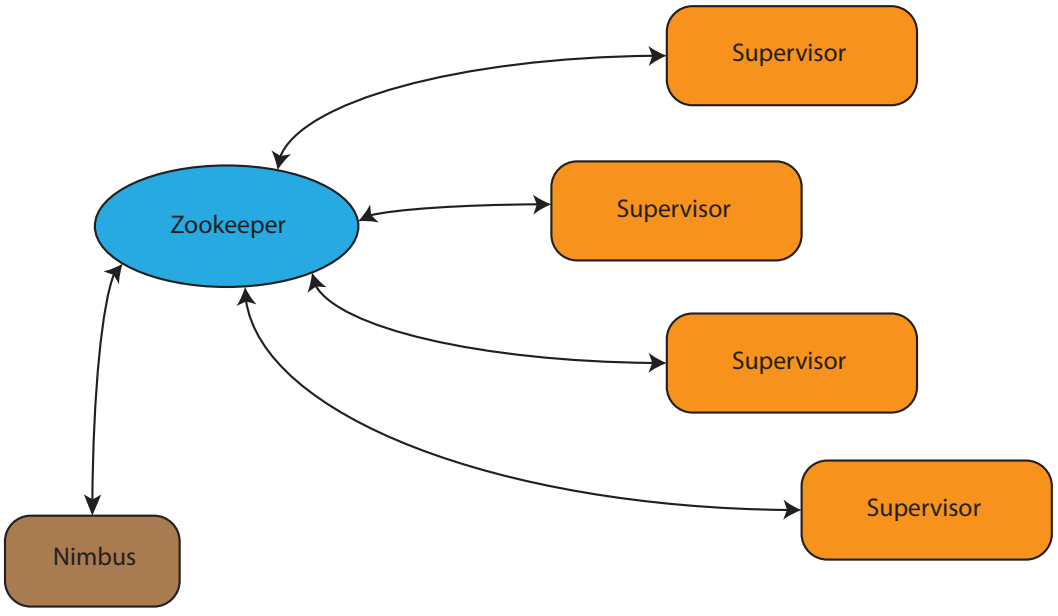
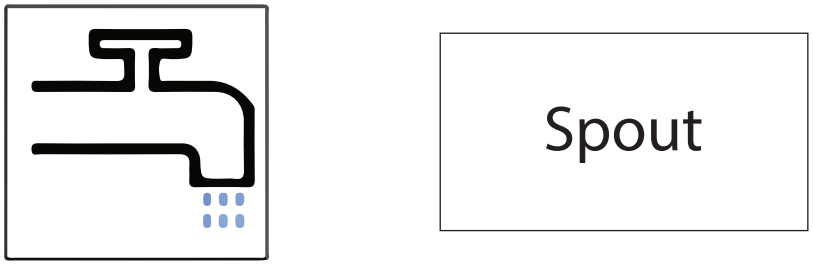


FIGURE 4.4: Storm Component Architecture

As we can see in the above figure the `Zookeeper` is responsible for the bi-directional communication and coordination between *Nimbus* and *Supervisor* daemons that reside in the *Master* and *Worker* nodes respectively.

4.2.4 Storm Stream Producers (spouts)

In `Storm`, `spouts` are one of the two basic primitives that the user at his disposal in order to perform the needed computations. A `spout` can be thought as a "tap" that produces tuples at a constant, variable or event driven rate; `spouts` usually read or "consume" data from external sources[1] and emit them into the `topology` for further processing. Depending on the user preferences a `spout` can be *reliable* or *unreliable*. This essentially means that a reliable `spout` is able to replay a tuple should its processing fails for any reason; in the *unreliable* case each tuple is sent once and then it is forgotten as soon it is emitted, regardless of its successful processing. Normally a `spout` is often depicted as a "tap" or a "rectangle box" in topology design drawings as it is shown in Figures 4.5a and 4.5b.



(A) Tap Representation (B) Box Representation

FIGURE 4.5: spout representations

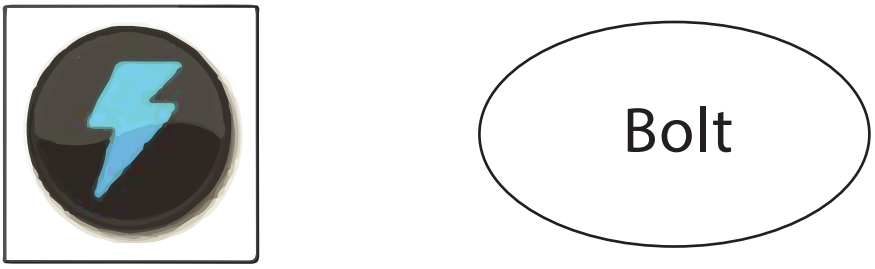
The default setting is that each `spout` will emit a single *output* stream that usually has the *default* id assigned to it; but this by no means a limitation as the user can declare at will any number of *output* streams from any `spout`.

4.2.5 Storm Stream Consumers (bolts)

The "workhorses" of each `topology` that is running in a `Storm` cluster is done using the second of the two primitives that are provided; they are called `bolts` and are responsible for all the transformations and processing that happens within each of the currently running topologies in the cluster. That includes for example tuple filtering, aggregation,

[1] such as `Apache Kafka`, Twitter `Kestrel` and others

joins, database interactions and a lot more. They are usually depicted in design drawings as a "lighting" or an ellipse as it is shown in Figure 4.6a and 4.6b.



(A) Thunder Representation

(B) Ellipse Representation

FIGURE 4.6: bolt representations

Bolts are designed as computation units that perform *lightweight* operations and if one needs to do complex processing or transformations then that functionality is recommended to be decomposed into multiple stages and hence, multiple `bolts`; one for each processing/transformation stage. Additionally, as was mentioned previously `bolts` in a similar fashion with `spouts` have initially one stream but the user can declare an arbitrary number of them.

Another differentiating characteristic when comparing `spouts` and `bolts` is that the former can only declare *output* streams while the latter can also declare *output* streams but is able to also subscribe to an arbitrary number of already declared streams these are called *input* streams and may be from different components, `bolts` or `spouts`.

4.2.6 Storm Topologies

A user of `Storm` that desires to perform any kind of processing or computation must first define a `topology` which is essentially a pre-configured graph of computation. Each vertex of the `topology` graph contains specific processing logic, defines the links that this node has and how the tuples should be passed around between the connected nodes. A `Storm` cluster can have a lot of `topologies` running at the same time while also performing the required load-balancing to have an even distribution of the available resources amongst them.

Mainly there are two distinct types of `topologies` in `Storm`; the *default* one that has high throughput but may process some tuples more than once and one that offers an *exactly-once* processing at the cost of lower-throughput; these `topologies` are called *Trident* topologies. Since `Storm` will process each tuple *at-least-once* (regardless of processing

failure) we can get that sort of processing guarantee in both types of `topologies`. What normal `topologies` can guarantee however is that each tuple in the `topology` graph is process *only-once*; that guarantee is a bit tricky to provide and `Storm` uses a lot of *acking* and state-keeping across the `topology`'s nodes in order to archive that; hence the extra communication cost overhead as well as the work required to support stateful nodes is what leads to the reduced performance when compared to regular `topologies`.

4.2.7 Storm Parallelism

`Storm` uses three distinct entities that are what actually run the `topology` in the cluster and we will now describe each one of them below.

- **Worker Processes**: A worker process executes a subset or subsets of the `topology` as a whole. Additionally a particular worker process is owned by exactly one `topology` and many run one or more executors (threads) for one or more `bolts` or `spouts` that belong to that `topology`.
- **Executors**: An executor is `Storm`'s notation for a thread that is spawned by a particular worker process; that thread many run one or more tasks of that `topology` (but usually we use one executor per task).
- **Tasks**: A task is that actually performs the actual data processing and manipulation; each of the `spout` or `bolt` implemented is executed by the use of many tasks that are spawned within executers at one or more worker processes that are started across the `Storm` cluster. It has to be noted that the amount of tasks remains the same throughout the lifetime of each running `topology`, but the number of *executors* for a particular component can change over time. That means that the following condition is true at any given time within each of the running `topologies`: number of executors $\leq$ number of tasks, but usually and as stated before we prefer to run one task per executor.

Figure 4.7 that follows outlines the internal structure of a worker process graphically and shows what runs where and with what analogy.

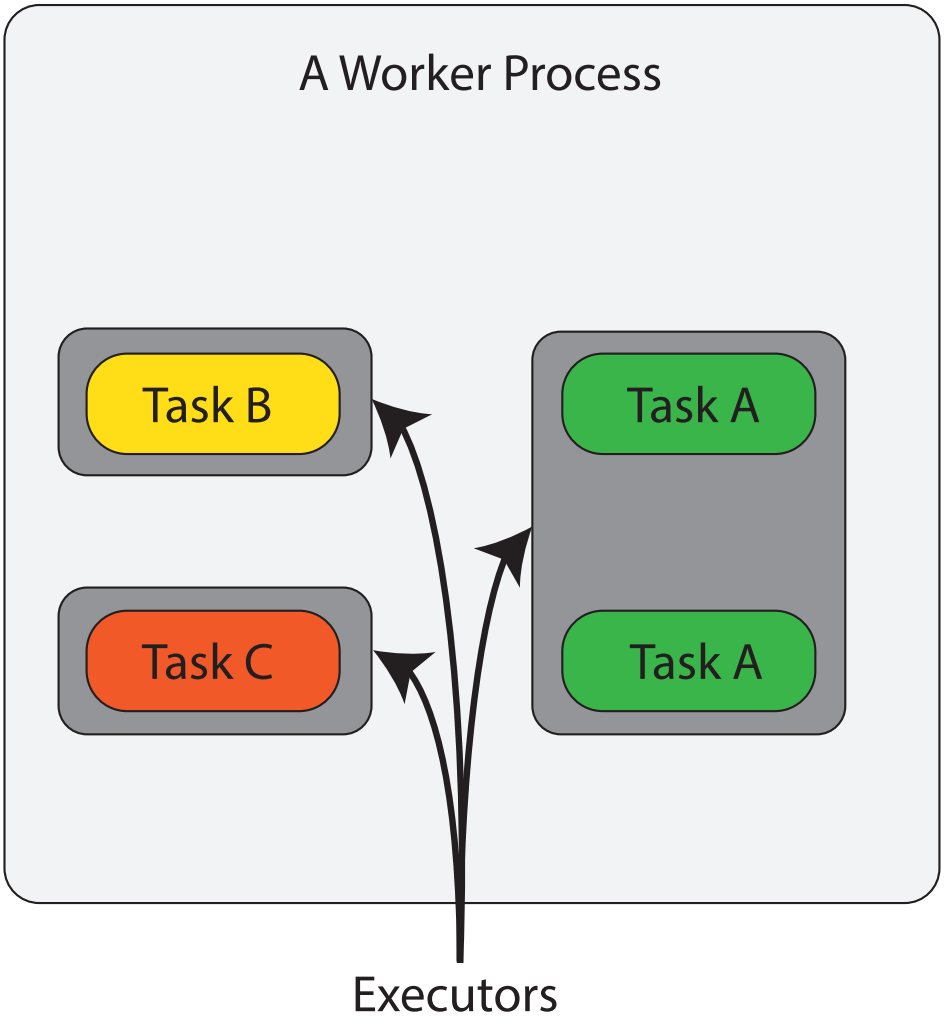


FIGURE 4.7: A sample worker process layout

Now that we have established the notions above we can now explain what parallelize options are provided by `Storm` for making the execution of `topologies` more efficient. First off, `Storm` uses a *parallelism-hint* that can be set individually for each of the primitives that are within a `topology`, this does *not* mean however, that we will spawn more `bolts` or `spouts` based on that hint; what it does mean is that it gives a suggestion to `Storm` on how many initial *executors* (threads) that primitive will have. Aside from the parallelism hint setting that is specific for each primitive, `Storm` gives us the means to configure the total number of worker processes that the topology can spawn across the cluster as well as the number of tasks per primitive.

Now let us assume that we have the `topology` execution graph that is shown in Figure 4.8.

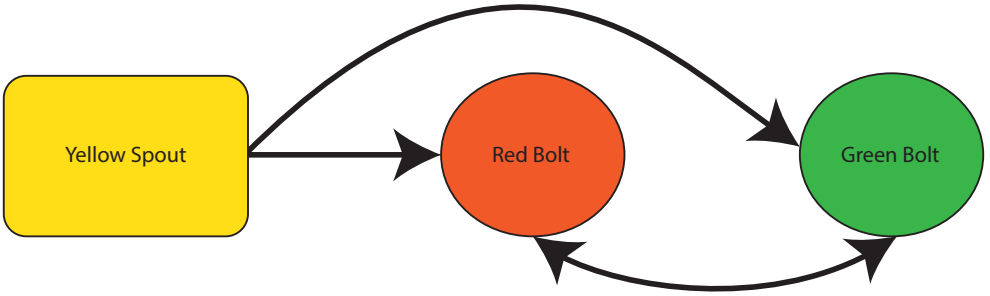


FIGURE 4.8: A sample topology graph

Additionally to the above let us assume that we have configured the `bolts` and `spouts` of that `topology` graph as follows:

- **Yellow Spout**: Parallelism Hint: 2, default number of tasks.
- **Red Bolt**: Parallelism Hint: 3, default number of tasks.
- **Green Bolt**: Parallelism Hint: 3, spawn 7 tasks.

Finally let us assume that we also instructed `Storm` programmatically to use 3 workers for the aforementioned `topology` graph. The graph above given the previously mentioned constrains would have a similar layout in its working processes as it is shown in Figure 4.9 that follows.

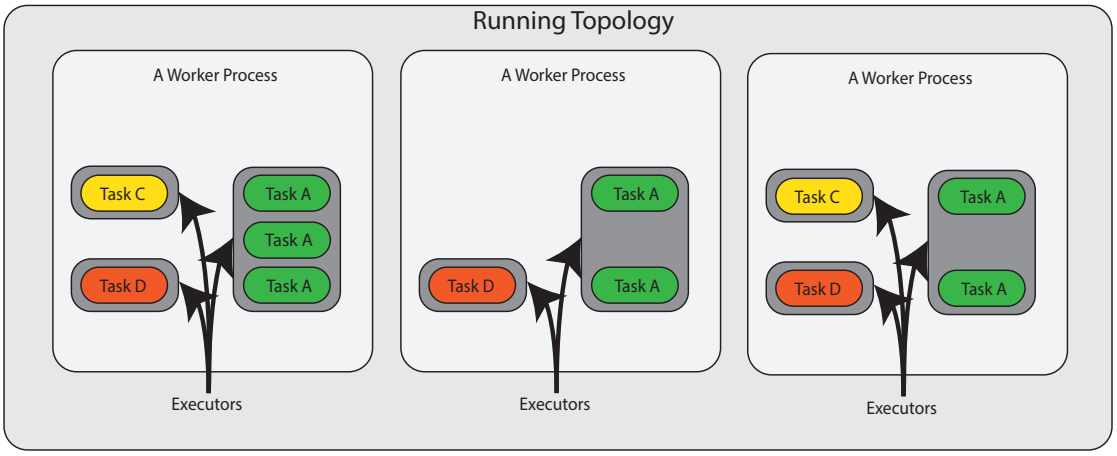


FIGURE 4.9: A sample running topology layout

It has to be noted that these mechanics are very useful due to the fact that `Storm` allows to perform *rebalancing* without taking the affected `topology` offline. This is a very handy

feature as one might image because it allows the `topology` to scale without changing its actual code or layout, hence avoiding its mandatory restart that these operations require.

The `topology` *rebalancing* can be performed using two ways, one via the command line and one through the web-interface provided by `Storm` itself. The settings for rebalancing have two different scopes, one for the whole `topology` where we can change the number of available worker processes that it has and the other scope is per primitive and allows us to change the number of its executors (i.e. threads).

4.2.8 Storm Grouping

`Storm` provides methods for grouping tuples using various ways which will be explained below. It has to be noted that these groupings should *not* be confused with actual routing; these groupings have only an effect within the same computation `bolt` and defines how the tuples are going to be distributed amongst the spawned *tasks* by that particular bolt.

- **Shuffle Grouping**: The most "usual" case, tuples are distributed as evenly as possible amongst the spawned `bolt` workers. Hence worker tasks will get, on average, the same amount of tuples for processing.
- **Fields Grouping**: The stream is partitioned based on its schema field(s) and as such the tuples that have equal key-fields will end up in the same task, while tuples with different are more likely to end up in different tasks.
- **All Grouping**: This grouping just sends to all spawned tasks a copy fo each received tuple, hence this grouping type has to be used with caution as it can generate a lot of extra, possible unneeded, work!
- **Global Grouping**: This particular grouping just indicates that the entire tuples of the stream are received by the `bolt` task that has the lowest *task-id*.
- **None Grouping**: This mean that we don't care how the stream is grouped; currently this grouping type is equal to *Shuffle-Grouping*. The eventual result though is, that `Storm` will push down `bolts` with none groupings to be executed in the same thread as the `bolt` or `spout` they subscribe from (when possible, i.e. they are only subscribed to one primitive).
- **Direct Grouping**: This particular grouping is a bit special; when a stream has this type of grouping for its tuples then the *producer* of the said tuples decides on which worker task of the *consumer* will receive this tuple. This to be enabled requires a direct stream declaration between the *producer* and *consumer* primitives.

The *producer* primitive can probe `Storm` at any time in order to receive the id's of its *consumer* tasks so that it can distribute its tuples accordingly.

- **Local or Shuffle Grouping**: Should the linked `bolt` has one or more worker tasks in the same process, then these tuples will be shuffled to just those in-process tasks. Otherwise, this grouping actually reverts to the default behavior that a normal shuffle grouping has.

4.3 Ruby On Rails

This is an *open-source* framework [55] that is written in `Ruby` [56] and provides a full-stack framework that is used to develop web applications that employ the Active-Record and Model-View-Controller design patterns. Using the provided tools and libraries we developed the front-end of our application that is used to show the processing results with ease. Additionally it provides scalability and adherence to recent web-standards with relatively no effort required. Hence due to the ease of use and features we selected this framework to develop the front end of our `Servicing-Layer`.

4.4 Discovering Patterns using `NeuralStream` framework

Since we require strong ordering and exactly one processing then we have to make use of `Storm`'s Trident API which enables the exactly once processing semantics. Unfortunately this comes at the cost of performance [2] as to provide the exactly once processing guarantee and strong ordering a lot more info needs to be exchanged inside the topology. Now in order to successfully implement `NeuralStream`'s local pattern extraction algorithm some design decisions had to be made.

First off in `NeuralStream` each batch has a fixed length but *not* height (which is the number of monitored streams by that group) for all of the groups regardless of the total number of monitored streams. We also assume that at each time tick something will be emitted from the spouts and in case of sensor inactivity/time-out then a value that indicates that fact is emitted instead or we substitute the missing values using the reconstruction $\hat{S}_i(t,:)$ at time t. We also must introduce the notion of *stream bucket granularity* which is necessary to avoid possible confusion in the system that might occur in some corner cases as we will later explain but first let's make an high-level illustration of how `NeuralStream` is actually structured; a basic example of a 2-Stream case is shown in Figure 4.10 that follows.

[2] when compared to regular non-Trident topologies

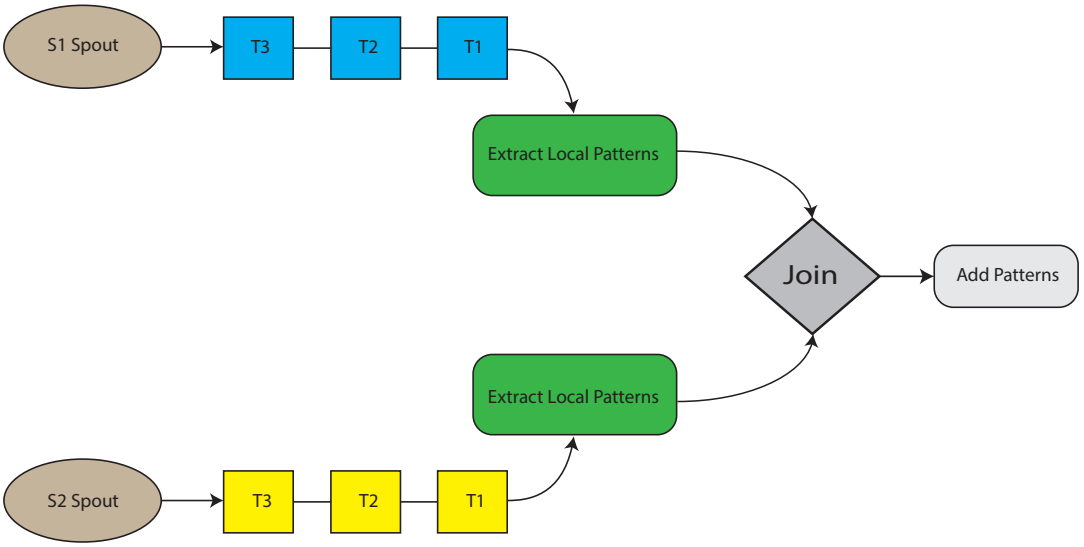


FIGURE 4.10: A 2-Stream `NeuralStream` overview

As we can see we have two `Spouts` each indicating a separate group, say S_1 and S_2, that emit tuples which contain the values of the monitored streams and an immutable timestamp indicating when these values were taken. Afterwards for each one of group tuples we perform the pattern extraction for all of the received tuples using Algorithm 5. The extracted patterns are then emitted and using the bucketization and joins are aggregated, joined and afterwards summed in order to produce the global patterns using Algorithm 6 for each time-bucket.

In the general case, we would have initially $S_i, i \in [1, m]$ number of groups that each one of them is assigned to monitor a number of streams. Each of the m groups performs the local pattern extraction and then forwards the result in order to calculate the global patterns as it was previously explained in detail within Chapter 3.

4.4.1 Stream Bucket Granularity

The need for granularity adjustments only presents itself in the multi-node scenario and occurs when we need to perform a *bucketization* operation on the time-measurements in order to be able to place them on the correct time bucket for processing. More specifically the problem occurs usually when we need to perform the *bucketization* operation in high emit frequency scenarios as there might be synchronization issues, not in the processing system itself but from the group clocks. To illustrate this problem let us first present the diagram in Figure 4.11 that follows.

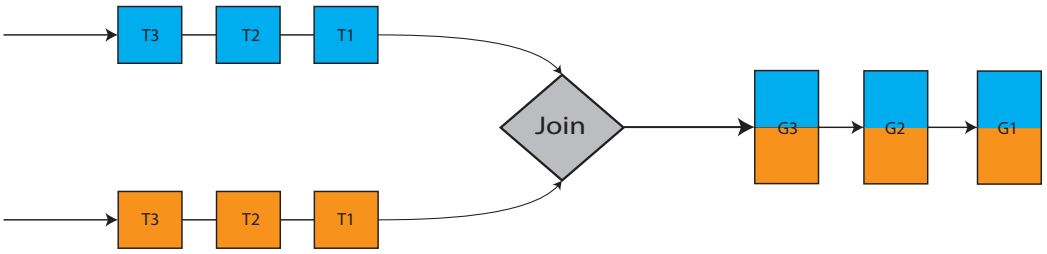


FIGURE 4.11: 2 Stream bucket join

In Figure 4.11 we see two tuple streams and let us assume that each one of the tuples transmitted are received in increasing order. The fields that each tuple has is a sensor value and a timestamp indicating when it was taken. When we perform the join operation it is normal to assume that the joins would be normally performed with a frequency equal to the tuple emit rate. Now the thing is, that this kind of discrete processing is costly and only possible if we have a tuple emit period that is large enough for it to be feasible (for example one tuple per second), but becomes *very* hard to perform in higher emit frequencies. This is due to the fact that we usually have some timer inconsistencies within each group emit timer thus causing a possible confusion. Enforcing strict timer synchronization is a sound solution albeit it becomes very hard to enforce when need to synchronize clocks of higher frequencies. A concrete example of the problem manifestation would be if node that executes a stream group $S_i, i \in [1, m]$ experiences a sudden CPU usage spike (that is out of our control) and processing is delayed by some milliseconds, hence the timestamp of that tuple will be incorrect (delayed). This of course *does-not* mean tuples are being lost or that *strong* ordering is not preserved as the tuples will both arrive and will be in the correct order but they will not have the expected timestamp so they will probably end up in the wrong timestamp bucket.

It has to be noted that is *not* a distributed systems challenge but an epistemological challenge as we need to enforce the domain knowledge in order to solve this problem. There are many solutions to this, such as enforcing a strict (timer) synchronization amongst all of the m groups, using incremental counters that are in-sync within each of the groups and others. All of the examined solutions in practice add quite a bit of unnecessary complexity to the system and are not providing the necessary performance and stability improvement that we would like; thus in the end we opted to implement a simple, yet elegant workaround for this problem using timestamps instead. Additionally to avoid problems when having to deal with different timezones timestamps are generated using the UTC timezone time and are parsed as such.

To solve this problem we have implemented a couple of solutions and the user depending on the trade-offs that need to be made can select the best one that suits the particular needs of his/her application as one of the two can be active at any given time throughout

the topology groups. The first solution is based on using *interpolation* to fill the missing values and the other one is to perform *averaging* of the slots within each bucket. Details on how we implement these techniques are explained later but first let us present in Figure 4.12 that follows an example of the aforementioned problem.

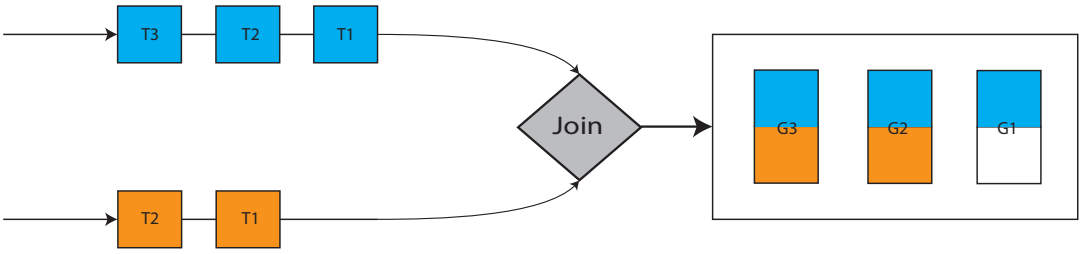


FIGURE 4.12: An incomplete 2 Stream bucket join

Here we see that the top stream has started emitting tuples slightly sooner than the bottom stream, hence in that time bucket after the bucketization operation the second stream will have one less tuple as it is shown above. Now using interpolation we would use it's nearest value in order to fill in the missing values (in this case the values from G2 orange tuple), thus we would have the following result as it is indicated in Figure 4.13 that follows.

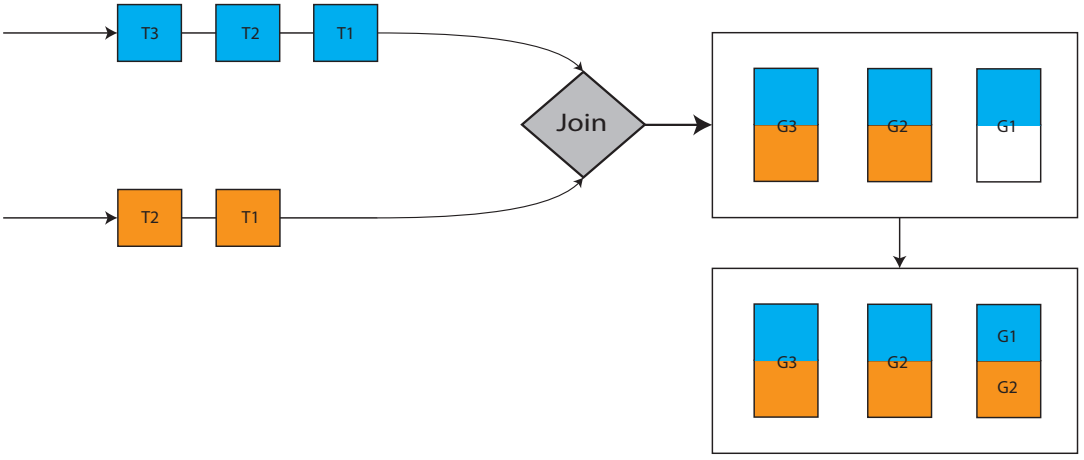


FIGURE 4.13: Fixing 2 Stream bucket join using interpolation

Now the other solution that we have built in to rectify the problem is to perform an averaging of the values in the time-bucket, effectively producing only one particular value for that time bucket and will probably "play-down" or smooth the intensity of

the extracted patterns which in some cases might be desirable. Figure 4.14 that follows shows an example of how the application of this method would look like in practice.

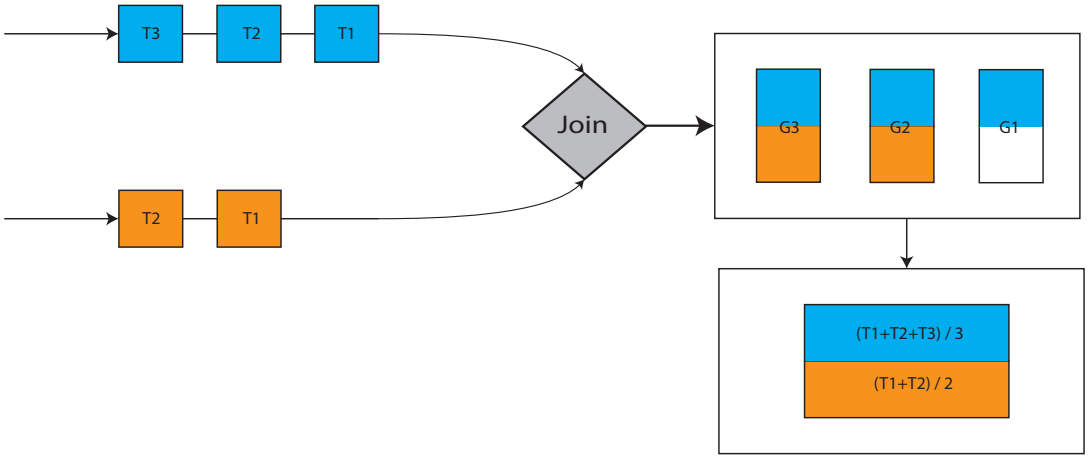


FIGURE 4.14: Fixing 2 Stream bucket join using value averaging

Since this is a solution that uses domain knowledge to resolve the aforementioned problem, we give the user of our system the ability to define a value for the *stream bucket granularity*, which shows how large the bucket buffer will be compared to the real tuple emit-rate. As we said previously this is only necessary when we have a very rapid tuple emit rate and is only used then. For example an acceptable bucket granularity if we emit 100 times per second (that is 1 tuple is emitted per 10ms) would be around the range of 250 or 100 ms and should be a multiple of the expected single tuple emit time (which in this case is 10 ms). As a rule of the thumb the granularity value should be a little higher than the time bucket you wish to have, e.g. a 100ms bucket would have a suitable granularity of $115 \approx 120$ ms.

4.4.2 Merging Extracted Patterns Efficiently

Using the *stream bucket granularity* notion as it was described in the previous section we can essentially guarantee that each of the generated time buckets will have a valid value. Each of the buckets will have a length equal to the granularity setting (e.g. 100 ms) and contain, depending on the method used to resolve the missing values (if any), one or multiple generated patterns from the group it originated; moreover, by design, all of the buckets from all groups will have the same number of values. This enables us to merge the local patterns in various ways with ease; this is attributed to the fact that these operations do not depend on the actual timestamp of the values but on a time-bucket that is linked to a particular time slot equal to the granularity value.

This enables us to perform merging with great flexibility; this is also aided by the fact that `Storm` topologies cannot be changed after creation and thus we need to "hardwire" its layout; although as it was previously noted we can perform a topology rebalance but this is not applicable due to the nature of our particular application. Naturally we have built into `NeuralStream` some basic features in order to provide a level of adjustment in the fan-out that we allow in the join stage, although it would be unreasonable to have more than one stage considering the network overhead but the option is still offered should one desire to change it. This factor essentially builds a balanced tree using that parameter. Two examples that use different balance factors for the join stages are depicted in Figures 4.15 and 4.16 that follow.

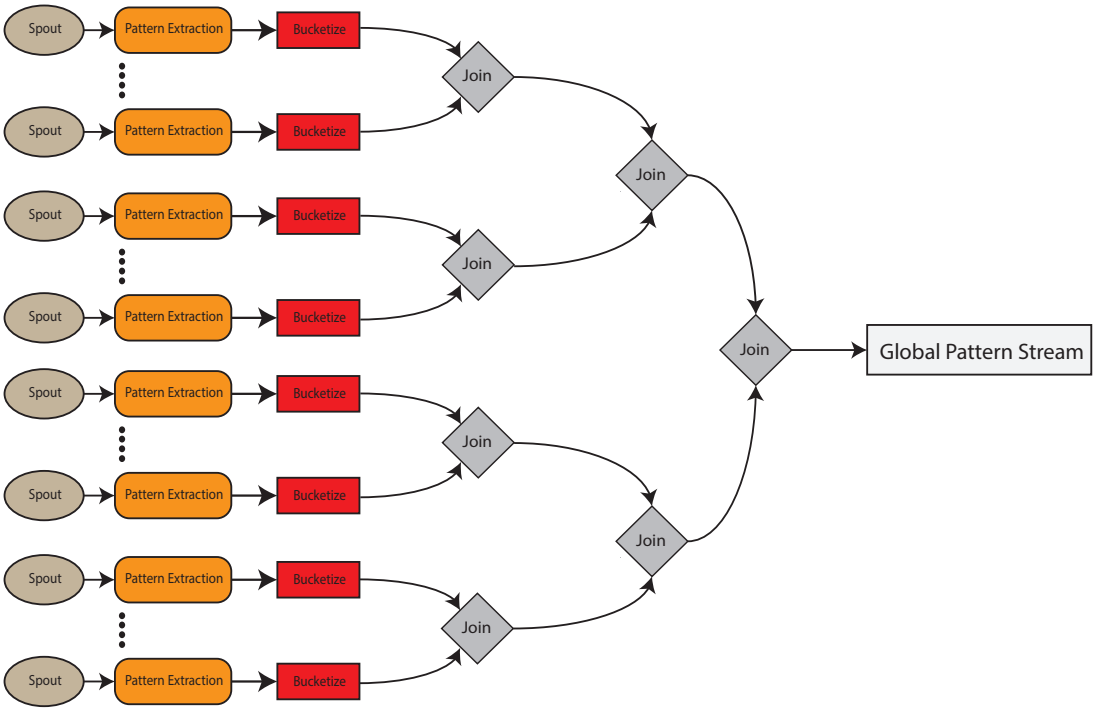


FIGURE 4.15: Example of pattern merging

The above balance factor is 2; now, a tree that has a different balance factor of 8 for the join stage is illustrated in Figure 4.16 that follows.

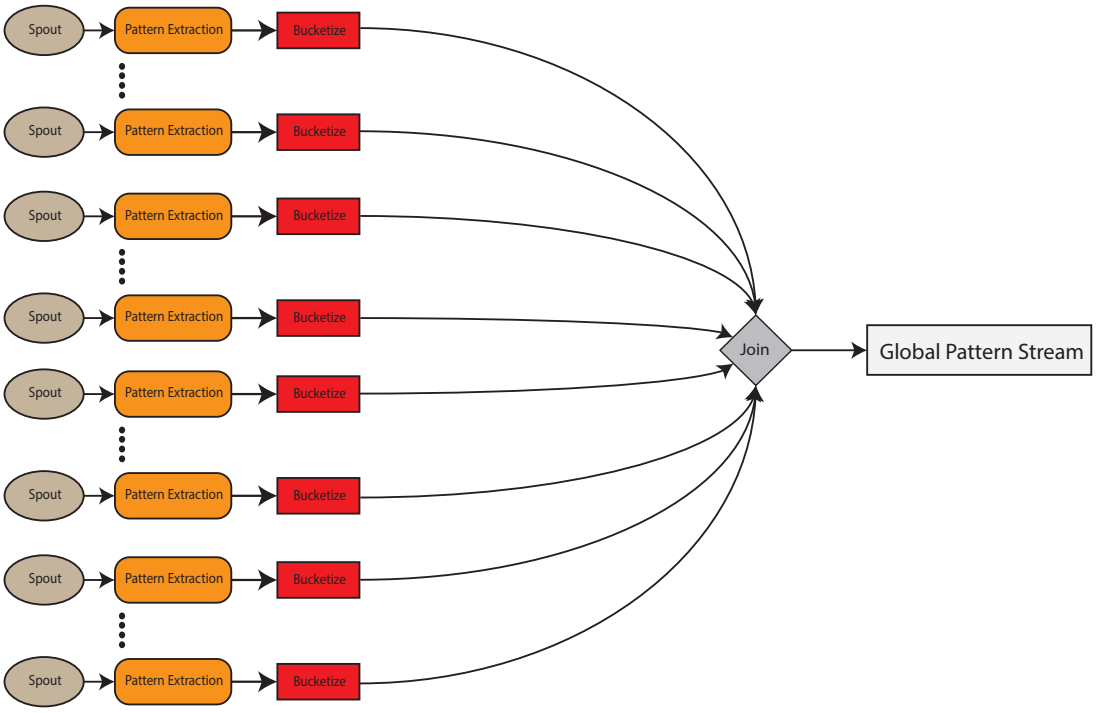


FIGURE 4.16: Example of pattern merging using a different balance factor

4.4.3 Further Optimizations

Generally in every stage, we try to perform as many optimizations as possible in order to increase the throughput of our framework and in this section we will outline some that are worth mentioning as they provide significant performance increases.

4.4.3.1 Optimizing Matrix operations

One of the worst performance hits and bottlenecks that you can have when you are executing a `Java` application is having to frequently invoke the garbage collector. This can cause huge delays in application responsiveness during the collection phase as memory is cycled from the `JVM` heap. The best way of (mostly) avoiding this problem is to preallocate the variables used in order to avoid for as long as possible the invocation of the garbage collector.

To better illustrate the problem let us use an example. We use two different methods of matrix generation before performing a transpose operation on a randomly generated matrix; the first one uses preallocation while the other one creates a new matrix for every trial. We run 100 trials for each point taking the average execution time for each one. It has to be noted that we use a High-Resolution-Timer (HRT) to measure the execution

time with an accuracy far greater than a *jiffy*[3]. The results are presented in Figure 4.17 that follows.

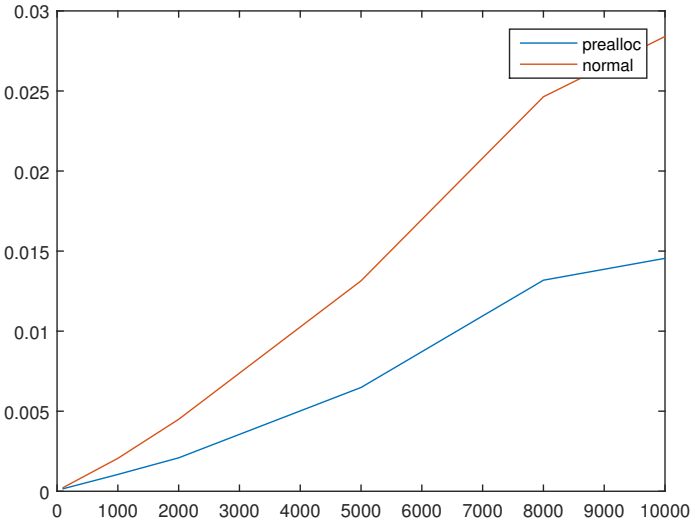


FIGURE 4.17: Transpose execution of a matrix with and without preallocation

We see that the performance impact between the two is dramatic, the effects are even more evident when we perform the most intensive method of our algorithm, the QR-decomposition that is performed at the end of each block. The results are shown in Figure 4.18 that follows.

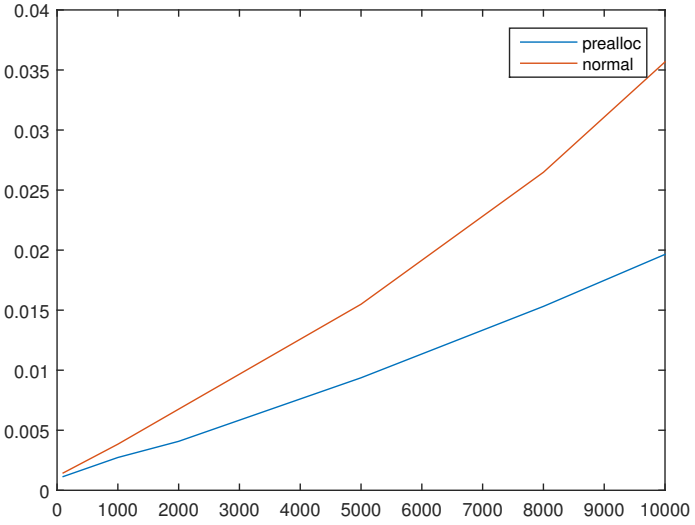


FIGURE 4.18: QR-Decomposition execution of a matrix with and without preallocation

[3]This is not a joke, it's an actual metric when you are not using a high-resolution timer; for more information please refer to Linux man pages

The results are even better than the previous one, the matrix that uses preallocation follows a linear curve as we increase the dimensionality while the normal one follows steeper curve which in the end tends to be exponential.

Now the question to ask is this: is it worth to have that performance-space trade-off? Well, to accurately answer that we have to first model how much space we expect to have allocated for each of our $S_i, i \in [1, m]$ groups that we will have in our `topology` and then we can accurately have an answer to that question based on the node specifications that we expect to have in our cluster. Let us have an example of the required space of two largest matrices that we need to keep, which are $S, Q \in \mathbb{R}^{p \times k}$ as they are shown in Algorithm 5 and give us an upper bound of the space we need[4]. In Figure 4.19 we can see an approximation of the storage required in order to store our tables in memory using preallocation.

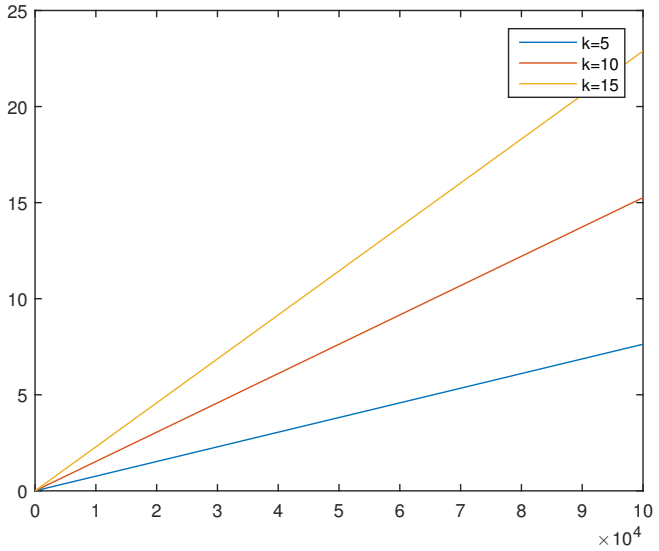


FIGURE 4.19: Approximate required storage (in Mb) for each S-group for dimension (x-axis) and k-patterns

As we can see, in the worst case that is shown in Figure 4.19 we require to track the top-15 patterns out of $100k$ monitored streams in each group. To achieve that we need only about 25Mb of memory for each of the m groups; couple that with the fact that RAM today is readily available in servers at much larger capacities than it used to, we think that in the majority of the cases we will be able to have that performance-space trade-off.

In practice, we give the user the ability to provide the maximum amount of desired patterns to be tracked at any given time within each of the m groups in the `topology`;

[4]Throughout our system we use double precision; hence each singular element requires 8-bytes (64-bits)

knowing that allows us to perform the preallocation of every vector or matrix that we will need as well as the generation of the p random disjoint variables. This preallocation results in a dramatic increase of throughput as it is indicated from the above graphs and is mainly responsible for maintaining the *online* processing for much higher dimensions.

4.4.3.2 Economy-Sized QR-Decomposition

Normally QR-decomposition requires a *full-rank* matrix, meaning that the resulting matrix would have to be square (e.g. $Q \in \mathbb{R}^{p \times p}$), but this is prohibitive for obvious reasons. As indicated in Algorithm 5 (as well as its previous iterations) we use a Q basis that is $\mathbb{R}^{p \times k}$, which depending on k can be considerably smaller in size than the squared one. This is due to the fact that we can perform QR-decomposition in "economy" mode that is sufficient in most cases and is obtained by solving using back-substitution the upper-triangular system shown in Equation 4.1.

$$R_1 x = Q_1^T b \tag{4.1}$$

This equation finds a suitable x that minimizes $||b - Ax||_2$ norm thus providing a solution to the least-squares problem, while keeping the size of Q_1 at the desired size $\mathbb{R}^{p \times k}$. Thankfully we only require the QR-factorization portion, so we do not have to "pay" in our case the cost of solving the least-squares problem that will find a suitable x. Our implementation uses `Apache Mahout` [57] math libraries for performing these operations since these are widely tested and have a solid numerical stability ensuring that our results will be accurate.

4.4.3.3 Optimal Stream Splitting

The splitting of the monitored streams amongst the available nodes is of utmost importance in order to archive a balance between scalability and throughput. Although `NeuralStream` offers the flexibility to have uneven number (dimensions) of monitored streams in each of the m groups it is highly discouraged to do so, as `Storm` will not be able to have an even computational load in each of the groups and as a result the load-balancing will probably suffer. Thus, as a basic suggestion (and as per `Storm`'s hints for performance) you should have 1 worker per node and an even (or very close to even) distribution of monitored streams in the m groups that the `topology` will have.

Chapter 5

Performance Evaluation

This chapter can be outlined into two main sections. The first section being the evaluation of the general algorithmic performance of our solution whilst performing a direct comparison with a very similar framework (`SPIRIT` [22]). The second part will be the actual performance evaluation of `NeuralStream` while it is being executed on an actual `Storm` cluster.

5.1 Algorithmic evaluation

Our algorithm as we have seen before in Chapter 3 is a combination of the algorithm presented by Mitliagkas [2] which provides the `PCA` approximation and subsequent basis to project onto while extracting the top-k patterns using the projection idea stemmed from Yang's [3] work.

Now we will delve a bit into the expected performance of our algorithm based on the theoretical bounds that are given in the Q_τ basis quality; we know from [2] that the subspace basis Q_τ after $\Omega(C_{\sigma,\lambda_k} p \log(p/\epsilon)/\epsilon^2)$[1] sample ticks will, with high probability, satisfy the following inequality $dist(Q_\tau, U) \leq \epsilon$. The *dist* function is the *largest-pricipal-angle-based* distance function between any two given subspaces of the same rank (as defined in [2]) and for completeness its definition follows in full.

Definition 1. Let $U, V \in \mathbb{R}^{p \times k}$ which represent the orthogonal basis of subspaces $span(U)$ and $span(V)$, respectively. Then, the distance between $span(U)$ and $span(V)$ is given by:

$$dist(span(U), span(V)) = dist(U, V) = ||U_\perp^\top V||_2 = ||V_\perp^\top U||_2 \tag{5.1}$$

[1] $C_{\sigma,\lambda_k} > 0$ constant and only depends on σ, λ_k

where $U_\perp$ and $V_\perp$ represent an orthogonal basis of the perpendicular subspace to $span(U)$ and $span(V)$ respectively.

For the finite sample analysis of the proposed `PCA` algorithm in [2] the authors assume a probabilistic generative model, from which the data is sampled at each step t; specifically each x_t is sampled from:

$$\mathbf{x}_t = A\mathbf{z}_t + \mathbf{w}_t \tag{5.2}$$

where $A \in \mathbb{R}^{p\times 1}$ is a fixed matrix, $\mathbf{z_t} \in \mathbb{R}^{k\times 1}$ is a multivariate normal random distribution variable; i.e.,

$$\mathbf{z}_t \sim \mathcal{N}(0_{k\times 1}, I_{k\times k}) \tag{5.3}$$

Finally $\mathbf{w}_t \in \mathbb{R}^{p\times 1}$ is the "noise" vector that is applied and is again sampled from a multivariate normal distribution such as,

$$\mathbf{w}_t \sim \mathcal{N}(0_{p\times 1}, \sigma^2 I_{p\times p}) \tag{5.4}$$

Now the complete theorem for ϵ-accurate recovery of the top-k principal components follows.

Theorem 5.1. *Let $\mathcal{X} = \{x_1, ..., x_n\}$ where $x_t \in \mathbb{R}^p$ for every t that is generated using Equation 5.2 and the SVD of $A \in \mathbb{R}^{p\times k}$ is given by $A = U\Lambda V^\top$. Let, without loss of generality, $\lambda_1 = 1 \geq \lambda_2 \geq ... \geq \lambda_k \geq 0$. Also let T be defined as follows:*

$$T = \Omega\left(\log(p/k\epsilon)/\log\left(\frac{\sigma^2 + 0.75\lambda_k^2}{\sigma^2 + 0.5\lambda_k^2}\right)\right) \tag{5.5}$$

Now let the block size be defined as follows:

$$B = \Omega\left(\frac{\left((1+\sigma)^2\sqrt{k} + \sigma\sqrt{1+\sigma^2}k\sqrt{p}\right)^2 \log(T)}{\lambda_k^4\epsilon^2}\right) \tag{5.6}$$

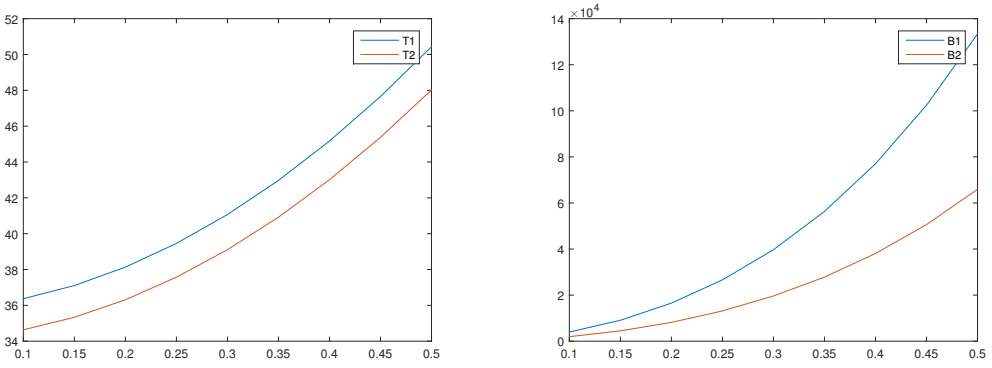
Then, after T block-updates, with probability 0.99 the inequality $dist(U, Q_\tau) \leq \epsilon$ will be satisfied. Hence, the sufficient number of samples to achieve an ϵ-accurate recovery of all the top-k principal components is:

$$N = \Omega\left(\frac{\left((1+\sigma)^2\sqrt{k} + \sigma\sqrt{1+\sigma^2}k\sqrt{p}\right)^2 \log\left(\log(p/k\epsilon)/\log\left(\frac{\sigma^2+0.75\lambda_k^2}{\sigma^2+0.5\lambda_k^2}\right)\right)}{\lambda_k^4\epsilon^2}\right) \tag{5.7}$$

Moreover we can suppress the extra log *factor that is placed upon T in favor of exposition clarity as T already appears in the expression linearly. Thus the above Equation 5.7 can be written as:*

$$N = \tilde{\Omega}\left(\frac{\left((1+\sigma)^2\sqrt{k} + \sigma\sqrt{1+\sigma^2}k\sqrt{p}\right)^2 \log(p/k\epsilon)}{\lambda_k^4\epsilon^2 \log\left(\frac{\sigma^2+0.75\lambda_k^2}{\sigma^2+0.5\lambda_k^2}\right)}\right) \tag{5.8}$$

For a detailed proof of this theorem one can refer to [2]. Now using the above equations for T, B and N respectively we can observe the following.



(A) T plot (B) Block size

FIGURE 5.1: T and Block size graphs for different σ (x-axis) and dimension

The T scales more linearly than the block itself and is very sensitive to the value of σ. It has to be noted that these are very extreme cases as the dimension (p) for the blue and red lines are 100000 and 50000 respectively; both graphs depict the results for the weakest principal component ($k = 10$ in both cases). Now we can see in Figure 5.2 that the required samples to recover the largest (and subsequently weakest) principal-component with ϵ-accuracy follows a pattern that is similar to the Block graph shown in Figure 5.1b.

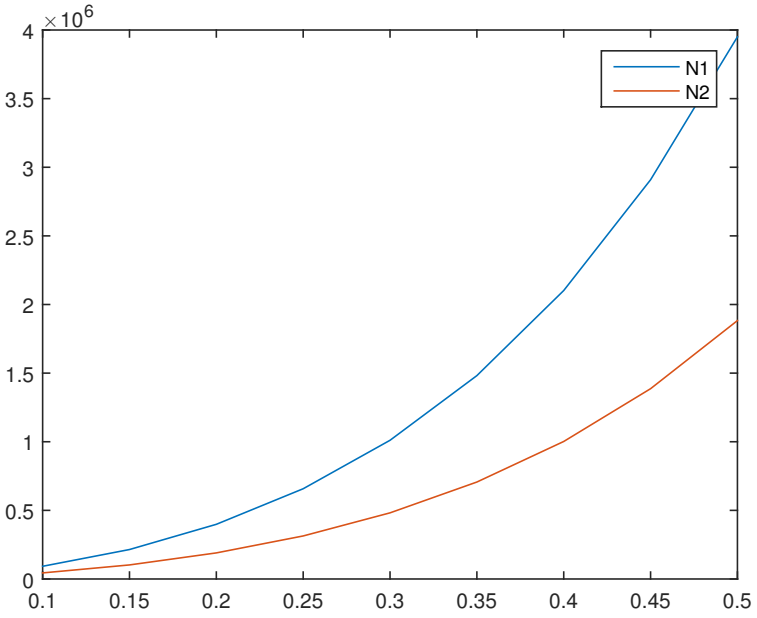
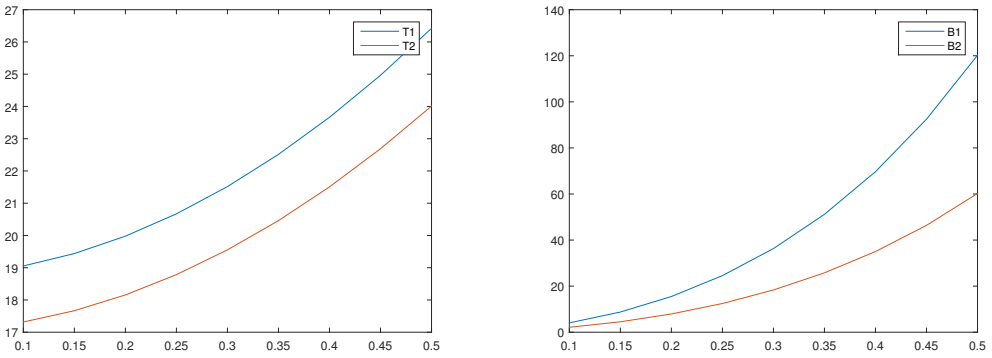


FIGURE 5.2: N samples required for ϵ-accurate recovery for $k = 10$

To see a more realistic scenario, let us use some reasonable p values, such as 1000, 500 respectively and keeping all other parameters the same. We now have the following graphs as shown in Figure 5.3.



(A) T plot (B) Block size

FIGURE 5.3: T and Block size graphs for different σ (x-axis) and dimension

The block size as we can see remains within a reasonable range as well as the number of block updates (T) that we require for an ϵ-accurate recovery. Now let us observe what happens to the number of samples in Figure 5.4, which again follows a similar pattern with respect to the block size as it is shown in Figure 5.3b.

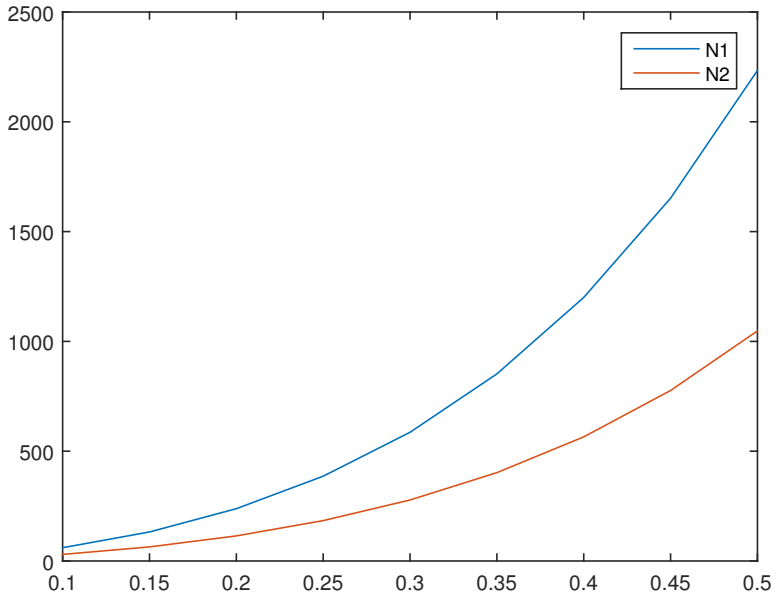


FIGURE 5.4: N samples required for ϵ-accurate recovery for $k = 10$

Thus, if we assume that the total variance within our time-series will be small, we can use the framework using a tiny block size while still being able to retrieve with the desired ϵ-accuracy all the principal-components. Interestingly enough, this is not an unreasonable assumption to make as it is indicated from our experiment results which we will see in the comparison graphs that follow.

5.2 Framework comparison

Now we will compare our algorithm and test its accuracy against another very popular and successful framework which was introduced in [22]. The aforementioned framework uses the complete PAST algorithm for projection approximation tracking outlined by Yang [3] for extracting the patterns out of multiple streaming time-series. In order to perform a similar and fair comparison we used the same datasets and compared the extracted pattern quality as well as the reconstruction quality across the datasets using objective metrics.

Additionally testing was performed on a second set of datasets as well; these datasets contain basically an extended version of the initial datasets (4 times the length of the original ones). These datasets were constructed by simply padding multiple initial datasets together thus creating a longer set of the same values and patterns. Interestingly enough, as will see from the benchmarks that follow `NeuralStream` is able to extract a greater

number of patterns from the dataset as well as achieve a better reconstruction quality in the long run across all of the datasets.

The following settings where used for `SPIRIT` throughout all the dataset executions (including both the regular as well as the expanded datasets),

- initial value of $k = 3$ (the default).
- Energy threshold was $[0.95, 0.98]$ (the default).
- exponential forgetting λ factor was 0.96.
- hold-off ticks was 100 (the default).

For `NeuralStream` we used the following settings again throughout all the dataset executions (including both the regular as well as the expanded datasets),

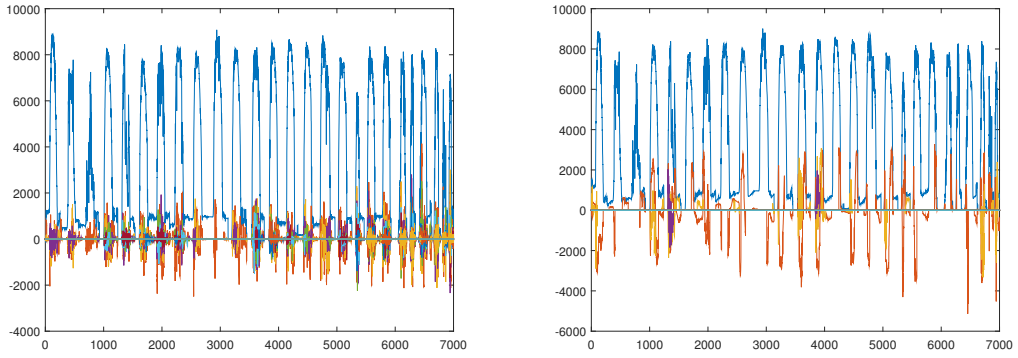
- initial value of $k = 3$.
- Energy threshold was $[0.96, 0.99]$.
- exponential forgetting λ factor was 0.96.
- block size was 6.
- hold-off was equal to block size.

It has to be noted that since the frameworks construct their `PCA` approximations in completely different ways the parameters above have different actual effects on each one of them and should not be assumed to be the same. Finally the metrics used for measuring the actual quality of the reconstruction is to measure the Peak SNR[2] observed throughout as well as the total mean-square-error (MSE) of the reconstruction. Finally in each table we will display after each dataset execution the energy ratio of the reconstruction against the actual data that is captured at that time from both frameworks.

[2]SNR: Signal-to-Noise ratio

5.2.1 Light data

The first dataset, contains light measurements from sensors; we see that this dataset has a lot of patterns throughout its duration. `NeuralStream`'s pattern extraction for this dataset is shown in Figure 5.5a while `SPIRIT`'s pattern extraction for the same dataset is shown in Figure 5.5b. As we can see `NeuralStream` is able to accurately detect a greater number of patterns when compared against `SPIRIT`; especially ones that have *low-energy*.



(A) `NeuralStream` patterns (B) `SPIRIT` patterns

FIGURE 5.5: Patterns extracted from Light dataset

Now we compare and see the reconstruction quality of both methods against the original data; this combined plot is shown in Figure 5.6 that follows. `NeuralStream`'s reconstruction is drawn using a *red*-colored line, `SPIRIT` is drawn using a *green*-colored line and finally the actual data are drawn using a *blue*-colored line.

We see that `NeuralStream` can more accurately follow the actual data, and the reconstruction quality remains high; additionally especially in high-peaked noise scenarios `SPIRIT` can over-estimate the reconstruction. Now to objectively compare the two we include the measured error metrics in Table 5.1 that follows.

Reconstruction Quality Comparison for Light Dataset		
Metric	`NeuralStream`	`SPIRIT`
MSE	$1.166550e+04$	$2.874328e+04$
Peak SNR	$7.461772e+00$	$3.545440e+00$
Energy Ratio	95.94	95.03

TABLE 5.1: Reconstruction Quality comparison for Light dataset

Based on these metrics `NeuralStream` is marginally better to `SPIRIT` when comparing the MSE and the peak SNR is approximately two times better in `SPIRIT`. We will now present the results for the padded (larger) dataset and perform the same comparison.

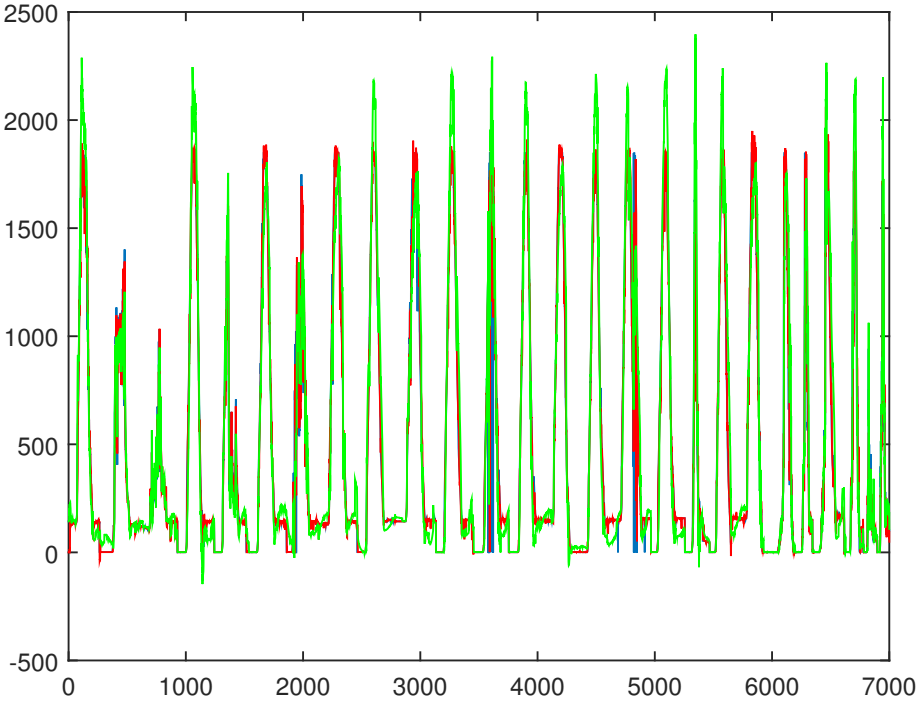


FIGURE 5.6: Light data reconstruction

Figure 5.7 shows the `NeuralStream`'s pattern extraction while Figure 5.8 shows `SPIRIT`'s pattern extraction for the same dataset.

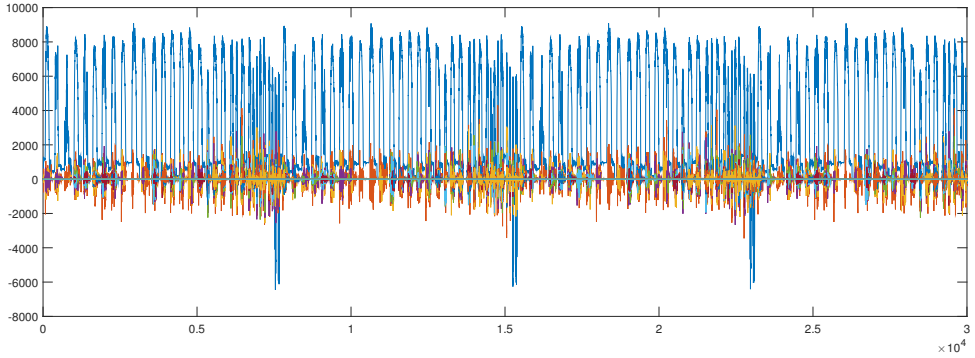


FIGURE 5.7: `NeuralStream` patterns in the larger dataset

As we can see the results are similar to the ones above, but now let's see the actual reconstruction for this dataset which is drawn using the same notation as previously[3] which is shown in Figure 5.9 that follows.

[3] `NeuralStream`: *red*-line, `SPIRIT`: *green*-line, Actual-data: *blue*-line

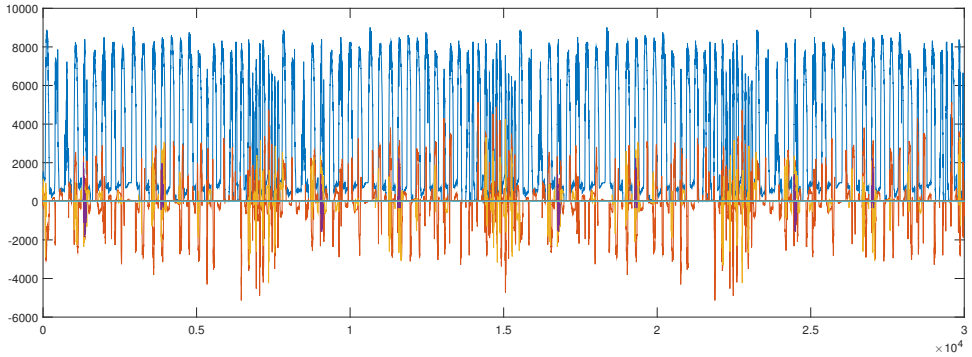


FIGURE 5.8: `SPIRIT` patterns in the larger dataset

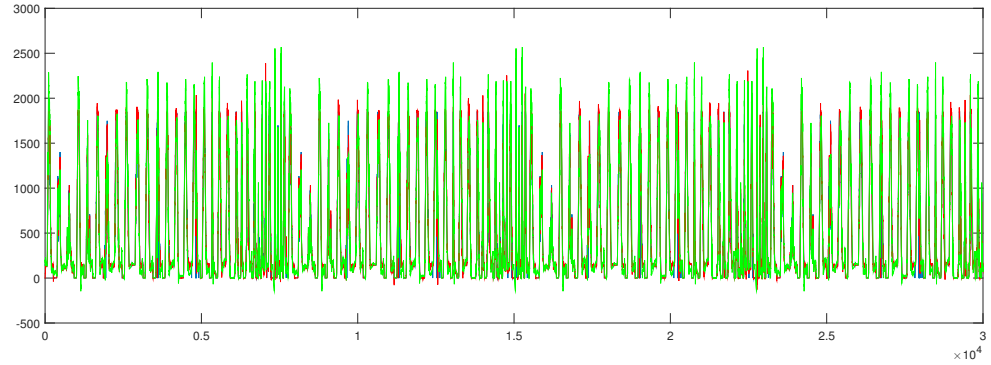


FIGURE 5.9: Light data reconstruction in the larger dataset

Again a similar graph as it was previously but it will be interesting to see how the quality metrics were affected by the increased dataset size; we display those results in Table 5.2 that follows.

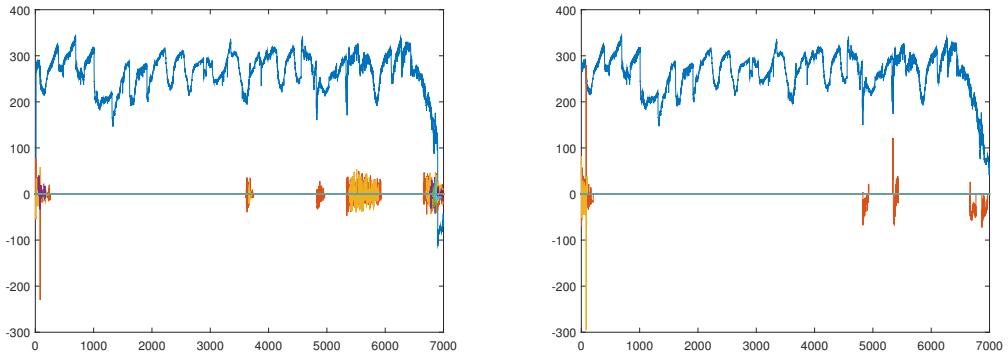
Reconstruction Quality Comparison for Light Dataset (4x)		
Metric	`NeuralStream`	`SPIRIT`
MSE	$1.145179e+04$	$2.858552e+04$
Peak SNR	$7.542068e+00$	$3.569343e+00$
Energy Ratio	96.17	95.03

TABLE 5.2: Reconstruction Quality comparison for Light dataset (4x)

As it is shown above the Peak SNR is slightly increased in both cases while the most important metric, the MSE, is further (slightly) reduced in both frameworks but as before `NeuralStream` substantially lower MSE shows that it again produced a marginally better reconstruction when compared to `SPIRIT`.

5.2.2 Humidity data

The second dataset, contains humidity measurements from sensors, again using the same notation as above Figure 5.10a shows `NeuralStream`'s pattern extraction while Figure 5.10b shows `SPIRIT`'s pattern extraction on the same dataset.



(A) `NeuralStream` patterns (B) `SPIRIT` patterns

FIGURE 5.10: Patterns extracted from Humidity dataset

We observe again that `NeuralStream` is able to capture more patterns, even if they have *low-energy*. Now let us graph the actual reconstruction for this dataset which is shown in Figure 5.11 that follows; using of course the same notation as before[4].

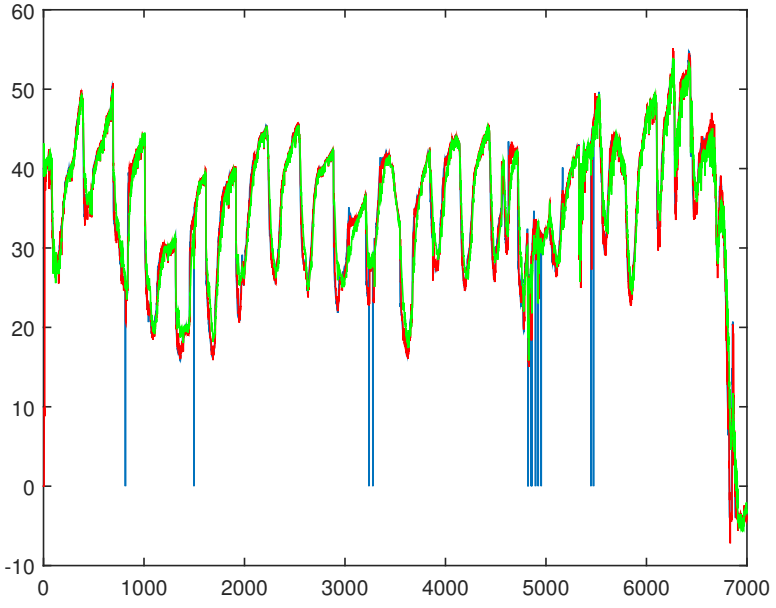


FIGURE 5.11: Humidity data reconstruction

[4] `NeuralStream`: *red*-line, `SPIRIT`: *green*-line, Actual-data: *blue*-line

Again `NeuralStream` is able to reconstruct the data more accurately than `SPIRIT` and this evident from the quality metrics that are shown in Table 5.3 that follows.

Reconstruction Quality Comparison for Humidity Dataset		
Metric	`NeuralStream`	`SPIRIT`
MSE	$1.922531e+01$	$1.943809e+01$
Peak SNR	$3.529207e+01$	$3.524427e+01$
Energy Ratio	96.27	106.73

TABLE 5.3: Reconstruction Quality comparison for Humidity dataset

As we can see from the table above `NeuralStream` is able to achieve a better reconstruction quality as its MSE is lower while the Peak SNR value remains close in both frameworks, albeit `SPIRIT` edges here a bit. Now we will observe what happens when we use the extended Humidity dataset instead. Figure 5.12 shows the `NeuralStream`'s pattern extraction while Figure 5.13 shows `SPIRIT`'s pattern extraction for the extended Humidity dataset.

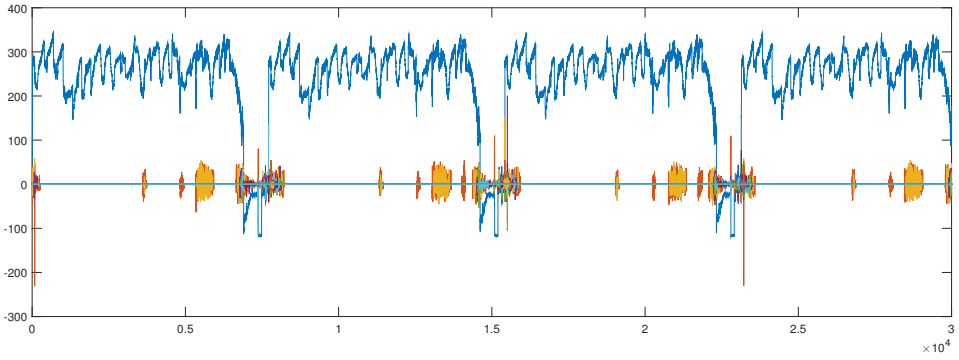


FIGURE 5.12: `NeuralStream` patterns in the larger dataset

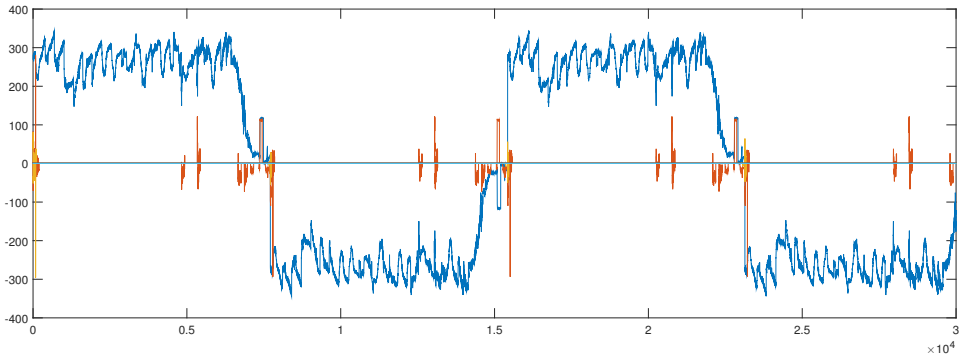


FIGURE 5.13: `SPIRIT` patterns in the larger dataset

The above graphs are a great example that showing `NeuralStream` ability at detecting patters with extremely high accuracy; for example the pattern that captures the most energy (drawn in both graphs with a blue line) has a lot of differences between `NeuralStream` and `SPIRIT`. But if we look at the original data reconstruction shown in Figure 5.14 the stream trend that is more evident resembles a lot more the pattern extracted by `NeuralStream` than the pattern that is extracted using `SPIRIT`.

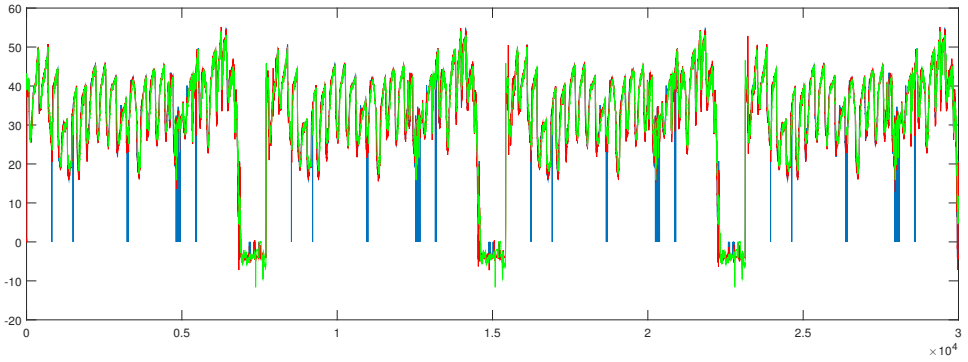


FIGURE 5.14: Humidity/Temperature data reconstruction in the larger dataset

Of course to back-up our claims we have to again compare the objective metrics which are presented in Table 5.4 that follows.

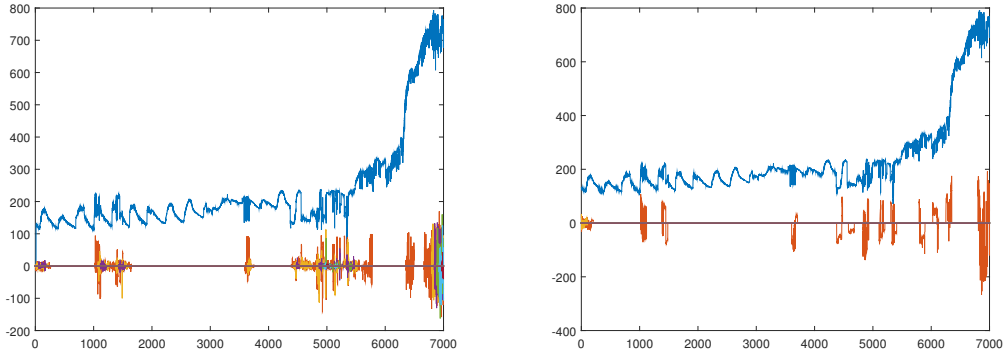
Reconstruction Quality Comparison for Humidity Dataset (4x)		
Metric	`NeuralStream`	`SPIRIT`
MSE	$1.759700e+01$	$1.960188e+01$
Peak SNR	$3.567642e+01$	$3.520783e+01$
Energy Ratio	96.28	98.30

TABLE 5.4: Reconstruction Quality comparison for Humidity dataset (4x)

As we can see the MSE of the reconstruction for the extended dataset of `NeuralStream` is considerably lower than the one produced by `SPIRIT` while the Peak SNR values remain close still backing our claims that the reconstruction quality is quite higher in `NeuralStream`.

5.2.3 Humidity-Temperature data

The third dataset, contains humidity/temperature measurements from sensors; using the same notation as above Figure 5.15a shows `NeuralStream`'s pattern extraction while Figure 5.15b shows `SPIRIT`'s pattern extraction on the regular Humidity/Temperature dataset.



(A) `NeuralStream` patterns (B) `SPIRIT` patterns

FIGURE 5.15: Patterns extracted from Humidity/Temperature dataset

Now the actual combined reconstruction graph for the Humidity/Temperature dataset is shown in Figure 5.16 using the same notation as in the previous graphs[5].

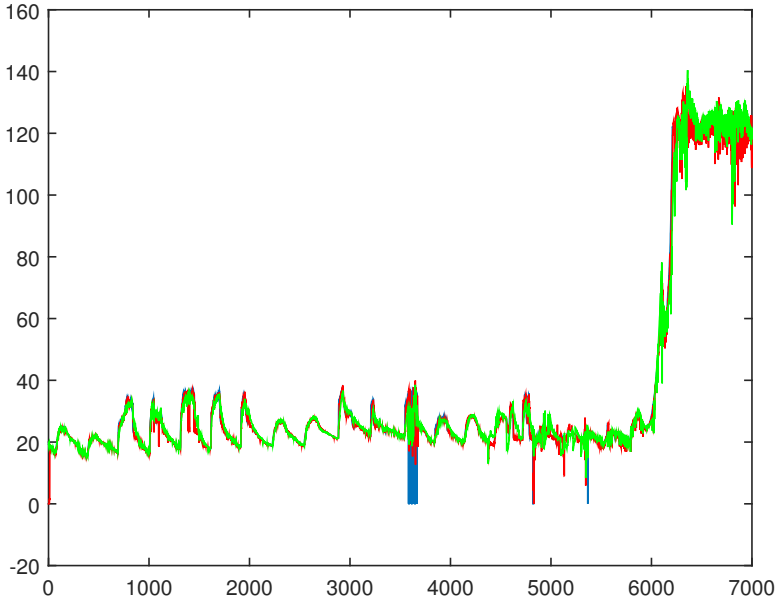


FIGURE 5.16: Humidity/Temperature data reconstruction

[5] `NeuralStream`: *red*-line, `SPIRIT`: *green*-line, Actual-data: *blue*-line

Similarly with the previous graphs `NeuralStream` is able to reconstruct the original values with grater accuracy than `SPIRIT`; this is also shown by the MSE and Peak SNR values of Table 5.5 that follows.

Reconstruction Quality Comparison for Temperature/Humidity Dataset		
Metric	`NeuralStream`	`SPIRIT`
MSE	$6.004453e+01$	$7.577711e+01$
Peak SNR	$3.034607e+01$	$2.933542e+01$
Energy Ratio	99.74	96.89

TABLE 5.5: Reconstruction Quality comparison for Temperature/Humidity dataset

Of course now we must apply both frameworks in the extended Humidity/Temperature dataset. Figure 5.17 shows the `NeuralStream`'s pattern extraction while Figure 5.18 shows `SPIRIT`'s pattern extraction for the extended Humidity/Temperature dataset.

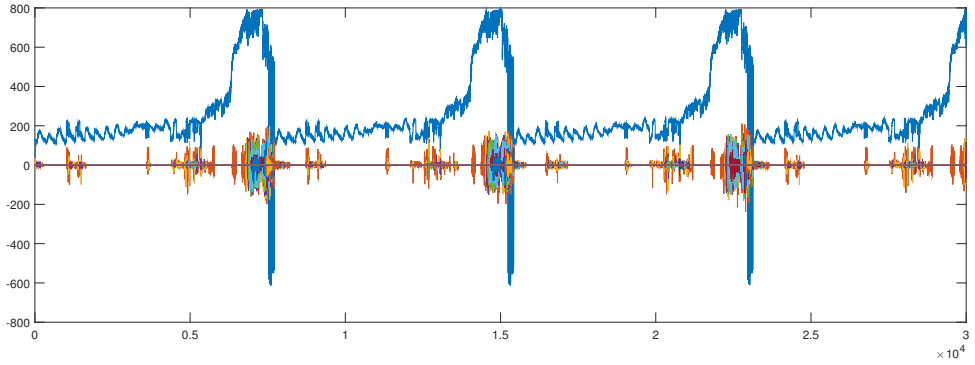


FIGURE 5.17: `NeuralStream` patterns in the larger dataset

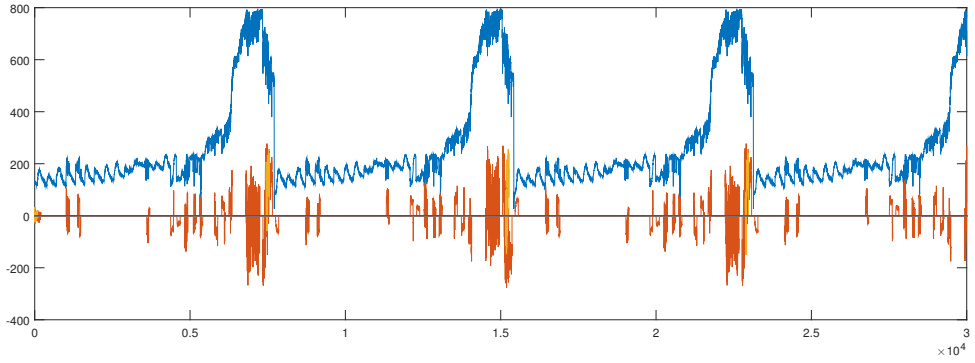


FIGURE 5.18: `SPIRIT` patterns in the larger dataset

The actual reconstruction for the extended Humidity/Temperature dataset is shown in Figure 5.19 that follows.

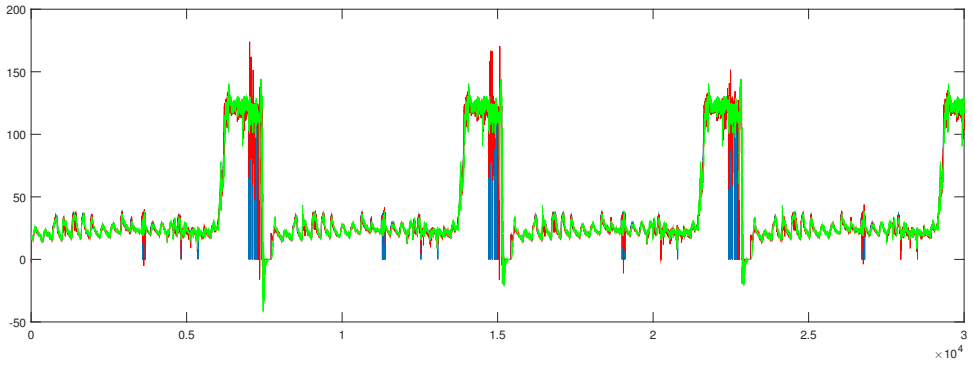


FIGURE 5.19: Humidity/Temperature data reconstruction in the larger dataset

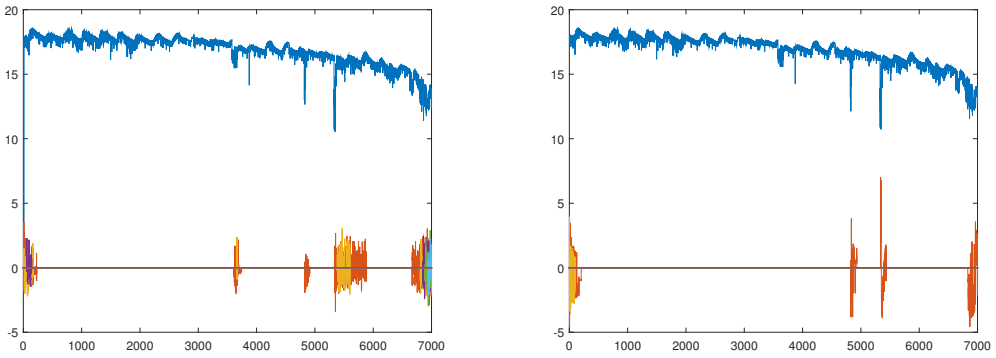
Although the reconstruction of `NeuralStream` quality again is significantly higher than `SPIRIT` in the extended set but, interestingly enough MSE is slightly reduced when compared to the regular dataset as it is shown in Table 5.6 that follows. It has also to be noted that this is the only extended dataset that MSE takes a slight hit; but as we previously stated the MSE remains marginally better when compared to `SPIRIT`.

Reconstruction Quality Comparison for Temperature/Humidity Dataset (4x)		
Metric	`NeuralStream`	`SPIRIT`
MSE	$6.092705e+01$	$7.745539e+01$
Peak SNR	$3.028270e+01$	$2.924029e+01$
Energy Ratio	99.76	96.89

TABLE 5.6: Reconstruction Quality comparison for Temperature/Humidity dataset (4x)

5.2.4 Voltage data

The fourth and final dataset, contains voltage measurements from sensors; using the same notation as above Figure 5.20a shows `NeuralStream`'s pattern extraction while Figure 5.20b shows `SPIRIT`'s pattern extraction on the regular Voltage dataset.



(A) `NeuralStream` patterns (B) `SPIRIT` patterns

FIGURE 5.20: Patterns from voltage dataset

Again `NeuralStream` is able to capture more patterns than `SPIRIT` can, but as we will see later on in Table 5.7 that follows the MSE for `NeuralStream`'s reconstruction is a little bit lower than `SPIRIT`'s. Now the actual reconstruction for this dataset through one run is the following:

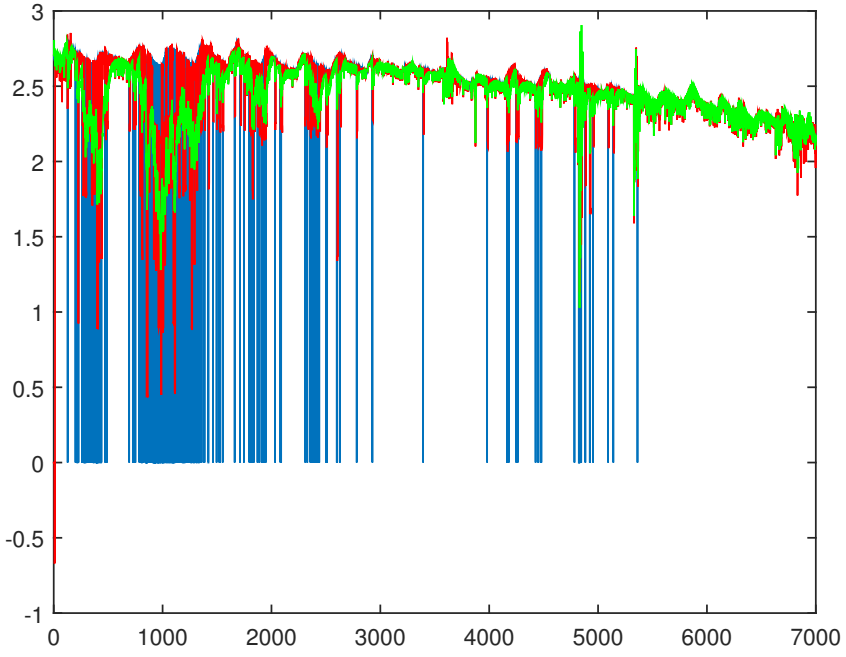


FIGURE 5.21: Voltage data reconstruction

Interestingly enough `NeuralStream` shows that is able to actually perform very decently following the actual data very closely; it has to be also noted that the MSE reported by both frameworks in this dataset is the *lowest* of all the other datasets reconstruction errors (and actually is very low).

Reconstruction Quality Comparison for Voltage Dataset		
Metric	`NeuralStream`	`SPIRIT`
MSE	$8.907373e-02$	$7.175533e-02$
Peak SNR	$5.863331e+01$	$5.957226e+01$
Energy Ratio	99.13	96.30

TABLE 5.7: Reconstruction Quality comparison for Voltage dataset

Now let us see the performance that both frameworks have in the extended dataset. Figure 5.22 shows the `NeuralStream`'s pattern extraction while Figure 5.23 shows `SPIRIT`'s pattern extraction for the extended voltage dataset.

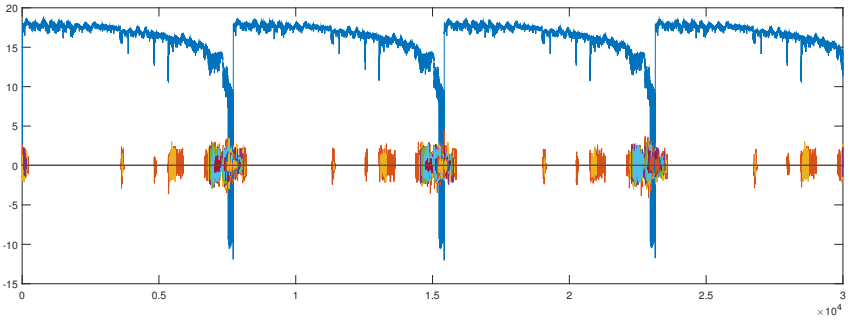


FIGURE 5.22: `NeuralStream` patterns in the larger dataset

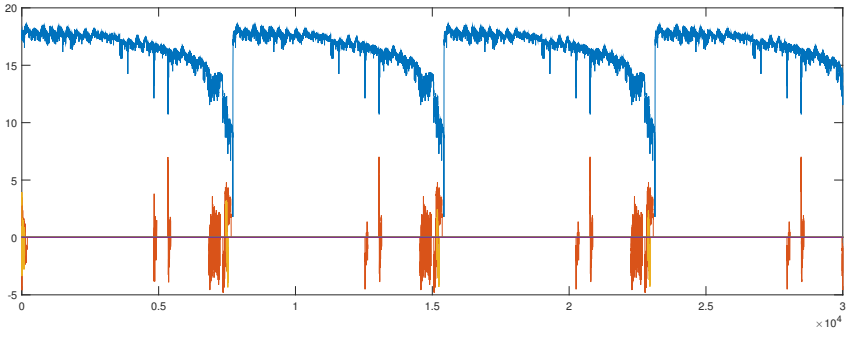


FIGURE 5.23: `SPIRIT` patterns in the larger dataset

Now the actual reconstruction for the extended voltage dataset is shown in Figure 5.24 that follows. It has to be noted that the notation used is the same as in the previous reconstruction graphs.

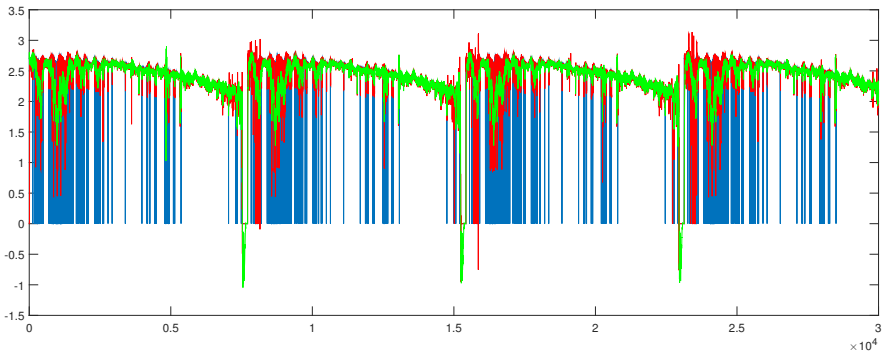


FIGURE 5.24: Voltage data reconstruction in the larger dataset

As we see from Table 5.8 the MSE of `NeuralStream`'s reconstruction suffers when compared to `SPIRIT`'s but the Peak SNR remains still favorable to `NeuralStream` albeit the differences are very small. Generally the MSE is very low for both frameworks and accurately capture enough energy.

Reconstruction Quality Comparison for Voltage Dataset (4x)		
Metric	`NeuralStream`	`SPIRIT`
MSE	$8.202285e-02$	$7.464013e-02$
Peak SNR	$5.899145e+01$	$5.940108e+01$
Energy Ratio	99.31	96.30

TABLE 5.8: Reconstruction Quality comparison for Voltage dataset (4x)

As a final note we find noteworthy that `NeuralStream` is able to capture the low-energy patterns that are missed by `SPIRIT` in all of the above cases. In the next section we will perform an evaluation of our framework when it is run on a `Storm` cluster.

5.3 `NeuralStream` cluster evaluation

In this section we will examine the actual performance of the multiple-node implementation of `NeuralStream` when it is executed on an actual `Storm` cluster. Before we present any of our results we have to have a solid expectation of what speedup to expect.

5.3.1 Amdahl's Law bound

Gene Amdahl made an argument in the seminal paper [58] that was presented in the AFIPS conference in 1967 and gave an expectation on the theoretical speedup that one can have given the parallel portion of the algorithm as a percentage. More specifically he stated that given the $n \in \mathbb{N}$ threads of execution and $B \in [0, 1]$ the fraction of the algorithm that is *strictly* serial then by using Amdahl's argument we can deduce that the approximate time $T(n)$ that an algorithm takes to finish when being executed on n number of threads is given by the Equation 5.9 that follows.

$$T(n) = T(1)\left(B + \frac{1}{n}(1 - B)\right) \tag{5.9}$$

Additionally given the formula presented in Equation 5.9 we can calculate the expected speedup of a program against the number of threads that it's going to be used for its execution using Equation 5.10.

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)\left(B + \frac{1}{n}(1 - B)\right)} = \frac{1}{B + \frac{1}{n}(1 - B)} \tag{5.10}$$

So, using Equation 5.10 that was shown above we can get an approximation on the expected speedup that we will get when trying to use more threads on the *same* machine. We can also extend Amdahl's argument into multiple-processors as well; so given the percentage of the *strictly* parallel portion of the algorithm as $P \in [0, 1]$ ($1 - P$ being the strictly non-parallel portion) and N the number of processors that will be used for

its execution the updated formula for calculating the expected speedup when using a multi-core scheme is given by Equation 5.11 that follows.

$$S(n) = \frac{1}{(1-P) + \frac{P}{N}} \tag{5.11}$$

Thus, using Equation 5.9 and Equation 5.11 we can now draw the desired curves in order to get an approximate speedup when using more computation resources for our algorithm. First we draw the speedup curves using Equation 5.9 which is shown in Figure 5.25 that follows.

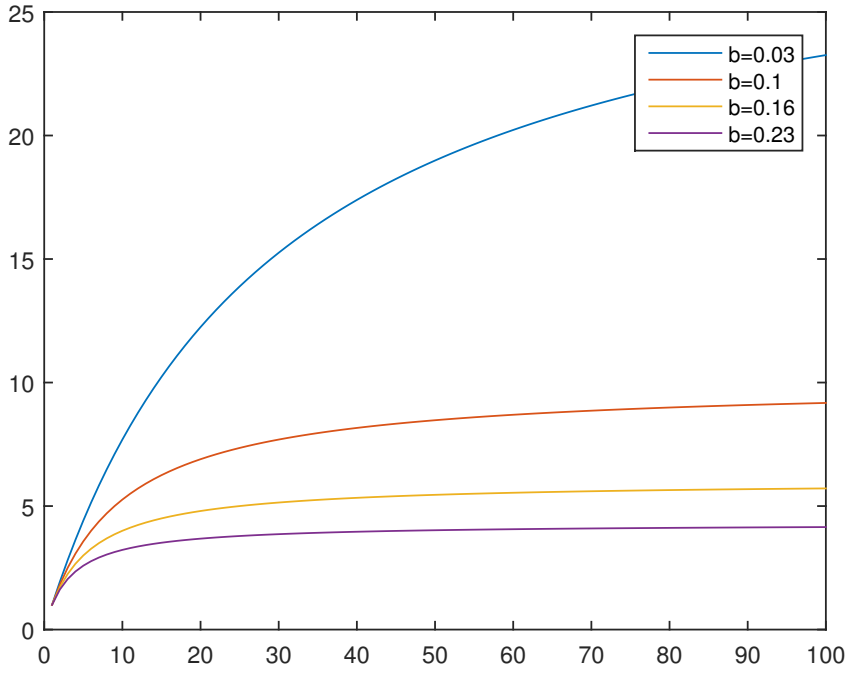


FIGURE 5.25: Amdahl's Law thread curves (x-axis n-threads, y-axis speedup factor)

We see that after one point using more threads yields limited or very small benefit unless the algorithm is massively parallel (e.g. *strictly* serial part b being ≤ 0.05). Unfortunately in the parallel version of the Amdahl's argument the benefit of using more processors diminishes even more quickly and this effect can be attributed to many things which

are out of the scope of this thesis; the speedup curves based on the parallel version of Amdahl's argument using Equation 5.11 are shown in Figure 5.26 that follows.

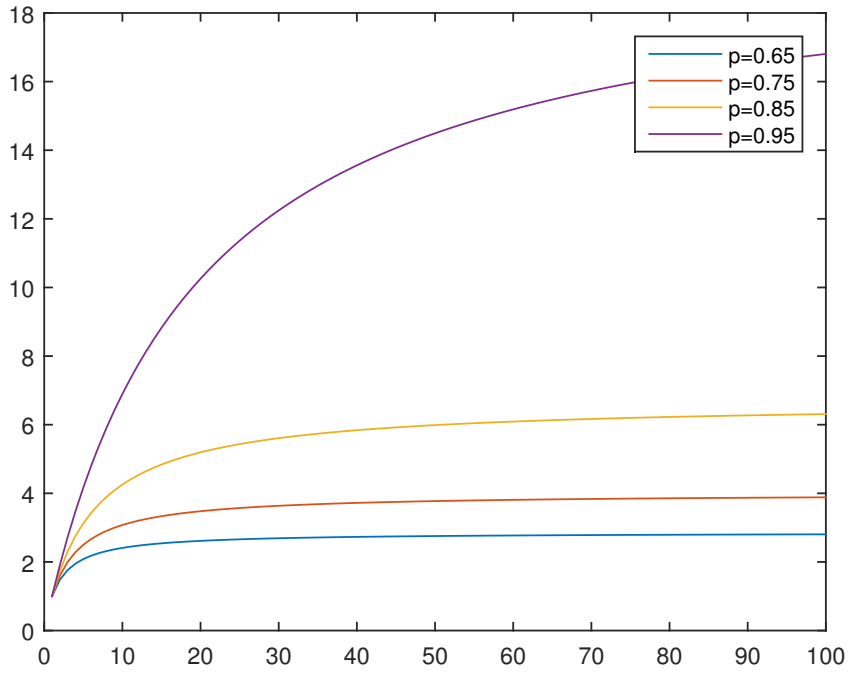


FIGURE 5.26: Amdahl's Law parallel curves

5.3.1.1 Limitations and Assumptions

However the Amdahl's law *does not* take in account the problem size changes and assumes that the problem will have a finite and fixed size; thus in our case we can apply Amdahl's law using the assumption that we will measure the expected speedup that we will have for each *time-tick*, as in our case each *time-tick* (which we define as the time required to process a tuple that initially belongs in $\mathbb{R}^{1\times p}$ from start to finish) has a finite and fixed size that remains, by design, approximately the same throughout our `topology` execution.

It has to be noted that Gustafson's law [59] as well as the Universal scalability law [60] are more modern and built upon Amdahl's law in order to better approximate the results; for the needs and purposes of the analysis performed in this thesis we find that the basic Amdahl's argument will suffice.

5.3.2 `NeuralStream` expected speedup

Firstly, to get a good estimation of the expected speedup and be able to fit it against the equations provided in the previous section need to model for each *time-tick* as a function; additionally we expect that each *time-tick* will take approximately the same resources to be completed (using the worst case each time). Hence a function that would be able to model each *time-tick* would be the following.

$$T_{ns} = Rtt_s + P_{proc} + Rtt_p + ceil(\frac{M}{J_{fanout}})(Rtt_j + J_{proc}) \tag{5.12}$$

Additionally, if we make the assumption that $Rtt_s = Rtt_p = Rtt_j$ and to be equal to $(Rtt_s + Rtt_p + Rtt_j)/3$ Equation 5.12 can further be transformed into the following:

$$T_{ns} = 2Rtt + P_{proc} + ceil\left(\frac{M}{J_{fanout}}\right)(Rtt + J_{proc}) \tag{5.13}$$

Where Rtt is the average *round-trip-time* at each stage and includes any data transfer required, P_{proc} the pattern extraction processing time required within each group, *ceil* is the ceiling function, M is the number of S groups in our `topology`, J_{proc} is the average time taken for each *join* operation as well as J_{fanout} which is the fanout setting for the *join* stages. We also assume the worst case regarding the number of Rtt's required, which is the one where each stage resides in a separate node.

The function is simple, and tries to model the actual processing stages as they are shown in Figure 4.15 and Figure 4.16. Each of the expected values is taken as an average, for example the Rtt value is the expected average of all Rtt's that occur in each tuple life-cycle. Of course the batching of each stage would have to be adjusted depending on the number of monitored streams for each group, it would be unreasonable to have a small batch when we monitor a relatively small amount of streams and vice-versa.

Now, using the function above we plot it using different values for the *Rtt* against the number of monitored streams per group, the results are shown in Figure 5.27 that follows.

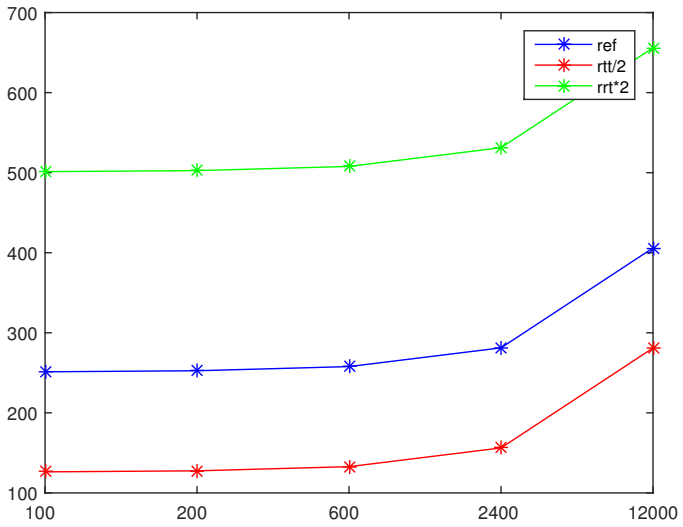


FIGURE 5.27: `NeuralStream` performance function *Rtt* sensitivity

We see that, understandably, the overall latency of our system depends a lot on the network performance but up to around 12000 streams per group we see that the latency remains under 1 second; which is the desired functionality if we want to keep the *online*-processing claim. Additionally let us see how other factors affect the scaling as well; in Figure 5.28 that follows we adjust the *join* stage fanout values as well as the total groups for the same amount of streams as before.

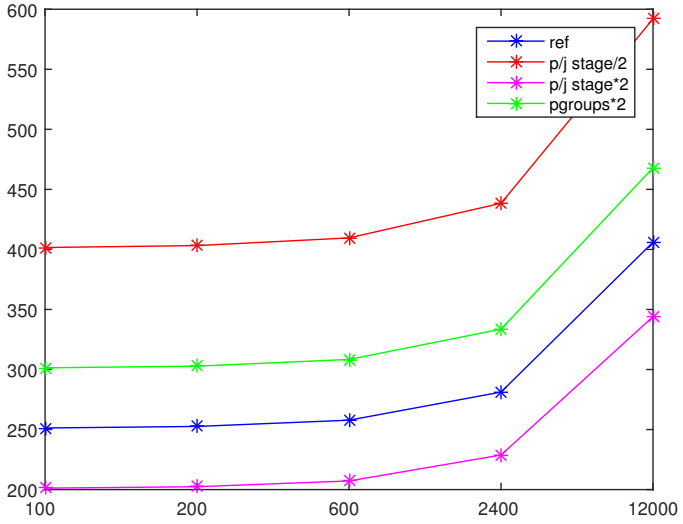


FIGURE 5.28: `NeuralStream` performance function sensitivity

As we can see from Figure 5.28 in both cases that use lower fanout values or more groups we end up practically increasing the amount of stages we have and hence the *Rtt*'s that have to "paid". In the other hand if we increase the allowed fanout value we see that we actually *reduce* the required stages and in turn the network transfer needed; thus, ending up with a better result overall.

5.3.3 `NeuralStream` real-world performance

Now using the findings of the previous sections we can analyze the real-world performance that we got when running `NeuralStream` on an actual `Storm` soft-cluster, which had the specifications shown in Table 5.9.

Soft-Cluster Configuration	
CPU	Core i7 3930K @ 4Ghrz (6-cores, 12 hardware threads)
RAM	8 × Corsair Dominator 4GB modules @ 1600Mhrz
Storage	2 × Samsung 840 Pro 256GB SSD in RAID0
Network	Hypervisor Ethernet Adapter (at least Gigabit)
Bare-metal OS	Ubuntu 14.10 Server

TABLE 5.9: `NeuralStream` soft-cluster system configuration

In order to simplify as well as standardize the soft-cluster creation we used `Wirbelsturm` which automates almost all of the hassle required for the node VM creation as well as their configuration; hence making executing `topologies` on soft-clusters and some cloud providers a breeze. Based on our hardware we used the configuration shown in Table 5.10 for each of the VM nodes generated and is not changed throughout our experiments.

Soft-Node Configuration	
CPU	1 vCPU with 2 vCores
RAM	3GB vRAM
Storage	10GB
Network	VirtualBox Network Adapter (at least Gigabit)
VM OS	CentOS
VM Hypervisor	VirtualBox

TABLE 5.10: Soft-Node configuration

Our target is to saturate the machine resources (or come close) trying to optimally extract patterns out of as many streams as possible using the hardware configuration of Table 5.9, while keeping the online-requirement. That basically means the total processing time

from start to finish for each *time-tick* has to be kept lower than 1000ms, which as shown in the previous section and Figures 5.27, 5.28 is expected to be around 12000 monitored streams per group; the goal now is to find how many groups can we effectively process before we start to lag. To find out we use four different populations of computation nodes in order to not only find out how many streams we can concurrently process but to evaluate the scalability of our system.

5.3.3.1 `NeuralStream` performance using 10 stream groups

In Figure 5.29 that follows we see the performance that we extract from our framework using 2,4,6 and 8 nodes while each one of them having the amount of streams in each iteration shown in the x-axis of Figure 5.29 while the y-axis represents the respective *time-tick* processing latency that we get for each run. Also *join* stage fanout was set at 4 and was equal for all configurations; additionally the maximum tracked patterns setting was set to 6 and was kept the same throughout our experiments.

We also had to adjust the number tuples in each batch depending on the of total monitored streams; more specifically we had to decrease it in order to avoid latency problems that would be introduced by having extremely large batches when we monitored a large amount of streams (e.g. ≥ 2000). The stream granularity setting was set to be equal to the batch emit rate, which was equal to 200ms at all cases.

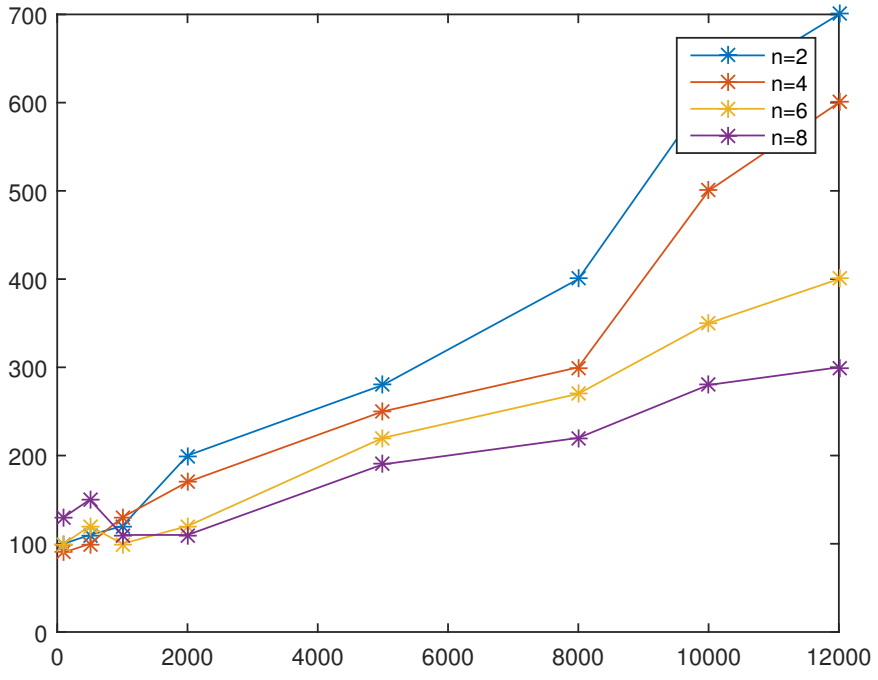


FIGURE 5.29: `NeuralStream` performance using 10 groups

Performance was sustained was within acceptable limits at all cases (i.e. was *online*); favoring in the end the most nodes at the most strenuous settings for this configuration (concurrently monitoring $120k$ streams amongst 10 stream groups).

5.3.3.2 `NeuralStream` performance using 20 stream groups

Since we were significantly below the 1000ms latency mark, we will now double the number of stream groups (20 in total) as well as the fanout value for both stages to 8 while keeping all the other parameters the same for each experiment.

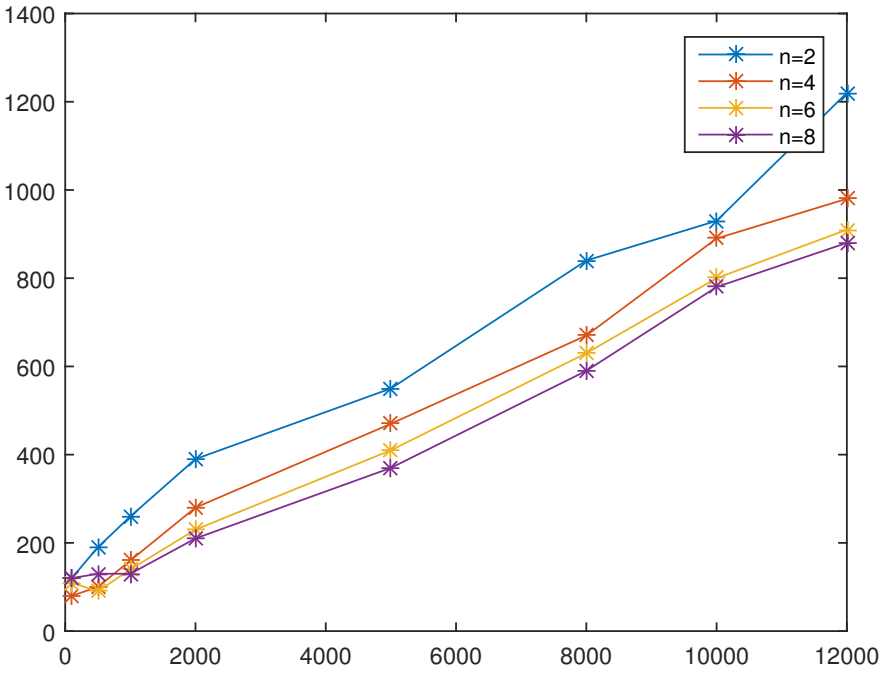


FIGURE 5.30: `NeuralStream` performance using 20 groups

In Figure 5.30 that is shown above it appears that we are starting to really saturate the hardware processing capabilities as we pass the 1000ms mark by a significant margin when we are using only two nodes at the most strenuous experiment, while all others are close to the 1000ms mark; additionally the gains are not as large when compared to the previous experiment (which is to be expected). Thus we can deduce that approximately we can monitor $240k$ streams in total using 8 VM nodes when using the hardware configuration of Table 5.9.

5.3.3.3 Performance Analysis

In order to quantify the scaling of our framework a bit better we shall now plot the scaling factors of the `NeuralStream` latency against the number of monitored streams in both cases; Figure 5.31 and Figure 5.32 show the latency scaling factors when using 10 and 20 stream groups respectively. The graphs show the factor that the overall processing latency has *decreased* (or if it is ≤ 0, increased) against the reference 2-node case. Figure 5.31 that shows the latency scaling for 10 stream groups follows.

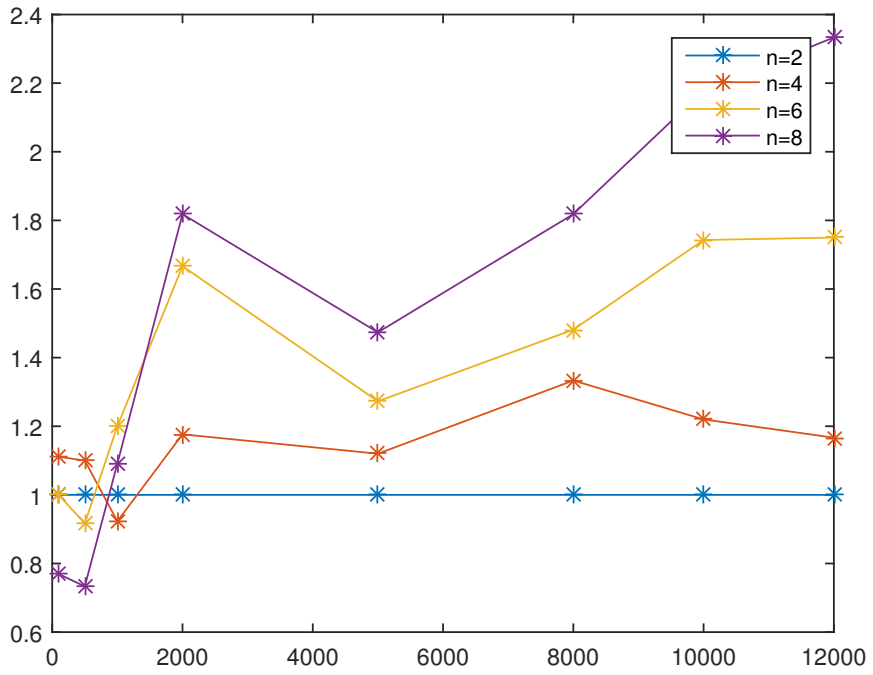


FIGURE 5.31: `NeuralStream` latency scaling when using 10 groups

Now we see that at best we can expect to have a latency reduction factor around 2.3x. As one might expect when we increase the dataset size and come close to the saturation point the returns (and hence the scaling factor) are diminished as Figure 5.32 that follows clearly shows.

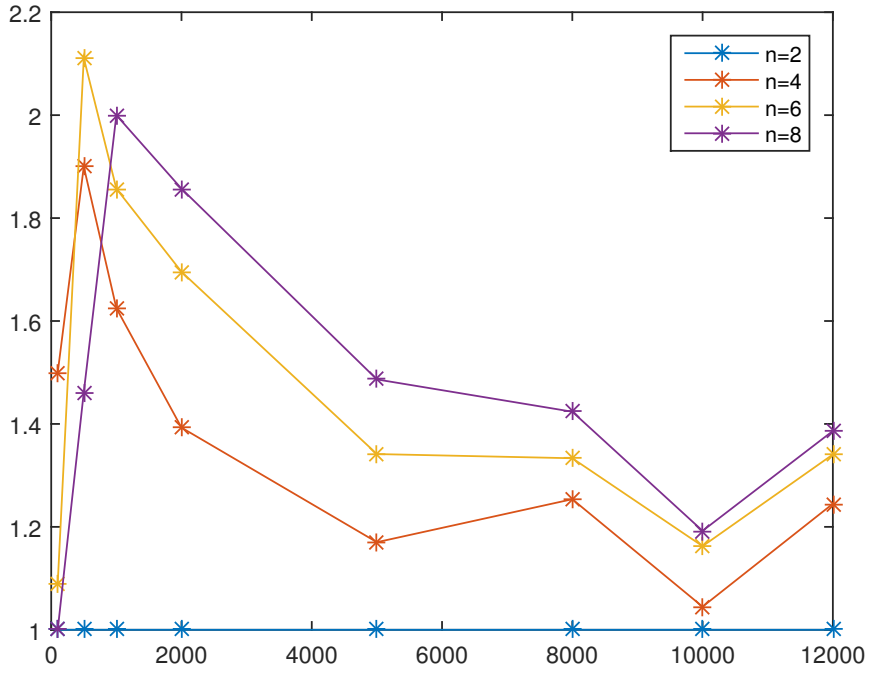


FIGURE 5.32: `NeuralStream` latency scaling when using 20 groups

Unfortunately the above figures seem that we are not getting very large speedup factors but in truth the speedup is actually very good and we shall now explain why. The scaling of the processing part against the maximum dimension of a stream group has nearly *linear* scaling as it was shown previously in Figure 4.18 [6] but sadly, the network speed and more importantly its *latency* does not. In fact the processing latency per tuple *increases* as we try to increase the total number of groups, stages or dimensions (the worst case of course being if you combine all three!); this happens not due to actual processing resource saturation but due to network restrictions.

Actually for monitoring thousands of streams we mandate the use of *at least* `Gigabit` Ethernet for that exact reason; if we fall-back to the order 100Mbit Ethernet standard then the theoretical processing throughput is cut by an order of magnitude, due to network restrictions as with 100Mbit one would be lucky to manage sustained network speeds of $\approx$ 9MB/s per NIC. On the other hand using `Gigabit` Ethernet we are able to cap ourselves (with an average quality NIC) around $\approx$ 85Mb/s per NIC; although, to achieve such (sustained) speeds one requires a *very* capable I/O subsystem (NAS-Arrays or SSD's) as well as a way to process network messages fast (which means either the NIC has hardware packet processing or the host computer has a beefy CPU). In fact if

[6] albeit it shows only for the QR-decomposition part, but generally the pattern extraction algorithm follows a linear scale-up curve; so this figure will suffice

we look back at Table 5.9 we use *two* SSD drives in RAID0 [7], which essentially doubles their performance; this was done in order to limit or eliminate in most cases the I/O subsystem bottleneck.

To better indicate the size of network transfer we will now show how the tuple size increase as we jump up the the monitored streams per group. Figure 5.33 shows the amount of space (in Megabytes) each tuple takes against the range of tested amount of streams per group.

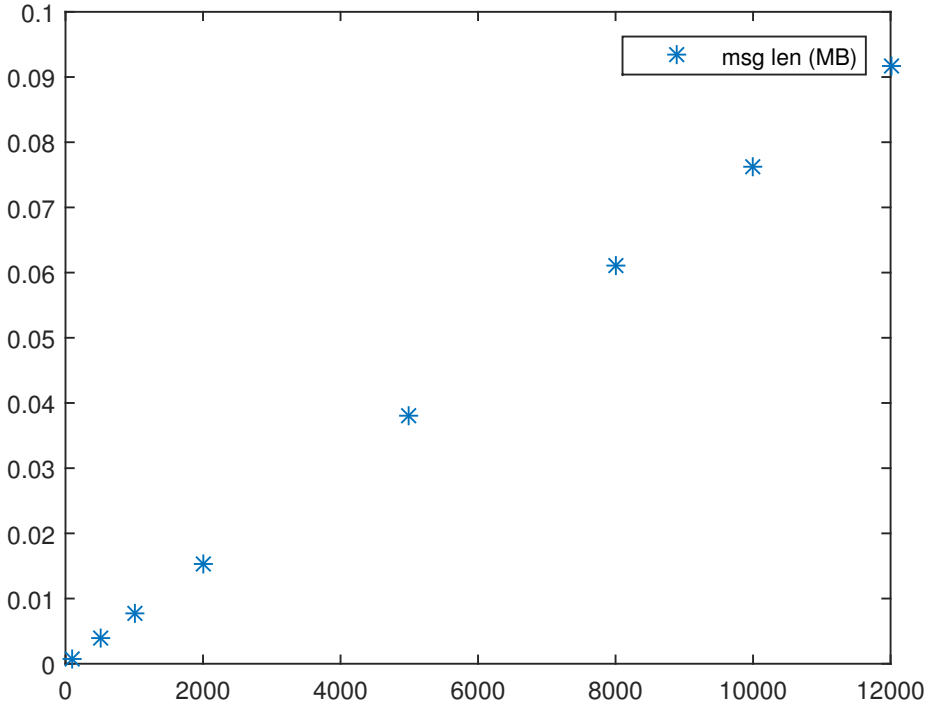


FIGURE 5.33: `NeuralStream` message size scaling for given dimensions

[7] In this configuration these drives can serve 200K IOPS and perform *sustained* Reads at ≈ 1.1GB/s and Writes at ≈ 1GB/s

Another important metric that we have to show is the amount of stress *stage-1*[8] puts to our network. Figure 5.34 shows the aggregated tuple size as well as the total traffic per second when using 10 and 20 groups respectively; for simplification we also assume that at all cases we emit 1 tuple per batch which is not always the case, especially when using lower dimensions.

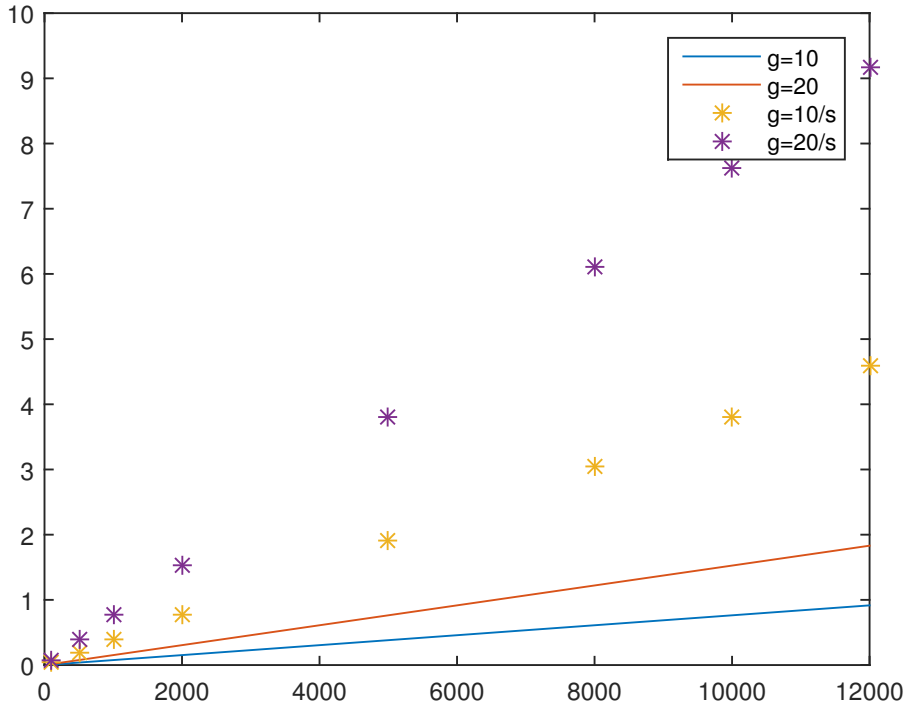


FIGURE 5.34: `NeuralStream` total (stage 1) message length scaling when using 10, 20 groups

Additionally since we perform a dimensionality reduction the *bucketization* and *join* stages on the order hand will be performed on tuples that belong in $\mathbb{R}^{1\times\max(k)}$ so compared to the *stage-1* transfer costs will be basically free. `Storm` is also exceptionally good at performing such aggregations when we have reduced the problem size to $\mathbb{R}^{1\times\max(k)}$ and thus happen extremely fast.

Another side benefit that is worth mentioning is that after *stage-1* the size of the produced tuples remains bound on the value of k regardless of the dimensions we have to monitor in each group. That fact *is* a feature and as is shown in Algorithm 6 tuple size depends only on the value of the maximum allowed patterns to be tracked in each group $(\max(k_i), i \in [1, m])$. As it was previously stated, it has to be noted that in the *Rtt* latency was taken as

[8]Stage-1 is defined as the stage from the Spouts → Pattern Extraction bolts

the average of all expected network latencies and *includes* the network transfers; further complicating the analysis by using separate *Rtt* values for each stage would yield slightly better results but at the cost of readability. For illustration purposes Figure 5.35 that follows shows where this bottleneck is.

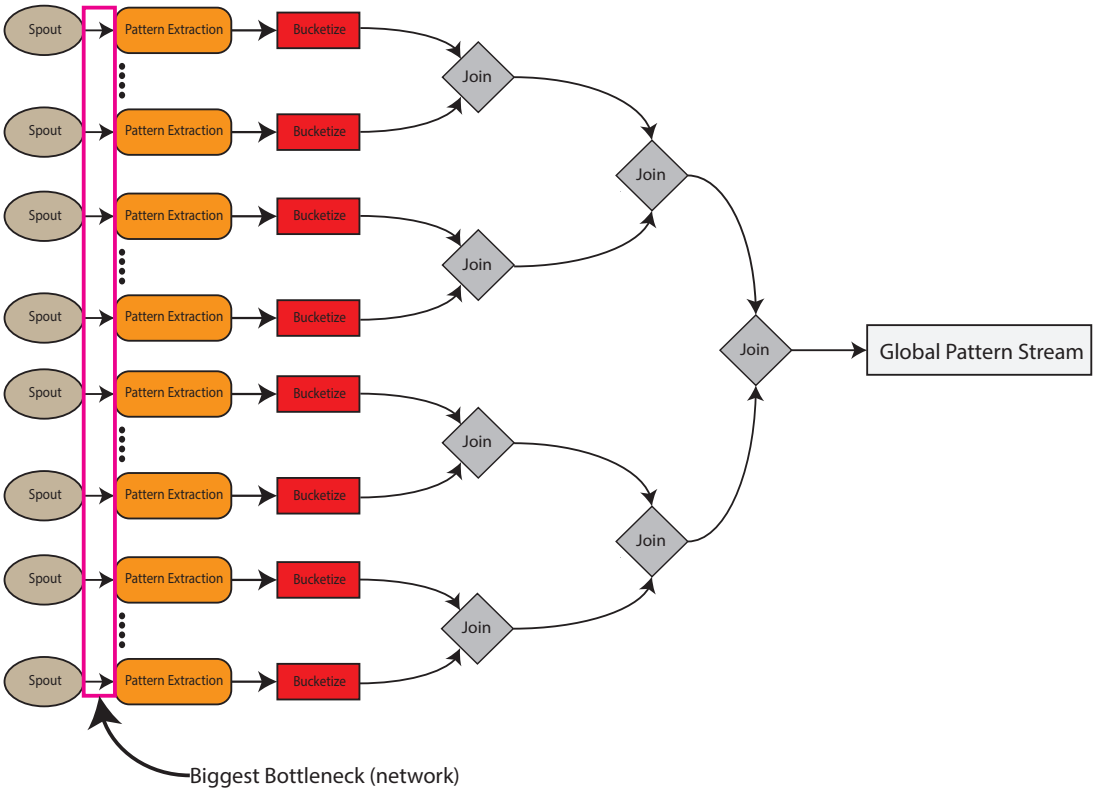


FIGURE 5.35: `NeuralStream` largest bottleneck

As a final note, we have to mention that the `topology` spouts in order to test the scalability of the system were assigned to emit a predefined and preallocated pattern which had the shape of a square pulse that was served in a way that saved as much memory as possible; this was what enabled us to scale out as much as we did because tuple generation is a very CPU intensive task; thus if generated the tuples completely in each spout then we would not be able to generate 5-tuples per second that had 12000 entries each. This was done in order to emulate the data being fetched outside of our system and *not* generated inside the spouts themselves, which would be the case in a production environment.

Chapter 6

Conclusion

In this thesis we tackled the problem of extracting representative patterns out of a vast amount of concurrent data streams that use the time-series representation model. Our experiments show that our algorithm is able to detect the top-k patterns out of n monitored streams with extremely high accuracy even if these have low energy, while having the lowest possible storage complexity $\mathrm{O}(kn)$. Furthermore scalability is unaffected as in the distributed version of our algorithm n is actually bound by the largest amount of streams in a group and *not* by the total amount of monitored streams. This novel approach is the heart of our `NeuralStream` pattern extraction framework which we introduce as an example use-case for our algorithm.

Natural expansion of this work would be to further refine the distributed version of the algorithm using an $E_{i,\tau}$ (i-th group at τ block update) correction matrix after the QR-decomposition step (which updates the projection base $Q_{i,\tau}$) within each group so that we can more accurately bias the local subspaces in order to ensure that the global projections converge at a much faster rate. Performing this in a way that will ensure the correctness of our algorithm without impacting performance is not a trivial task and requires significant effort.

Additionally, improvements to the usability of our pattern extraction framework would be another likely area of focus; with the most useful feature being the ability to provide pluggable spout implementations for fetching data from a reliable distributed queue (such as `Apache Kafka`). It has to be noted that although this feature does not require significant effort to be implemented it has been intentionally omitted. This is because `Kafka` at the moment [1] is undergoing a major consumer API redesign which will be released along with 0.9 version; hence, for the sake of future-proofing this feature should be delayed until the new consumer API is rectified and pushed into the stable channel

[1] at the time of writing the version on the stable channel was 0.8.2

(which will probably happen in the second or third quarter of 2015). Finally the front-end of the framework is provided as a proof-of-concept and there are a lot of usability improvements that can be made should one wants to use it in a production environment.

Bibliography

[1] N. Marz et al. Storm Processor: A Distributed and fault-tolerant realtime computation framework. https://storm.apache.com/, 2013-.

[2] Ioannis Mitliagkas, Constantine Caramanis, and Prateek Jainm. Streaming, memory-limited pca. 2013.

[3] Bin Yang. Projection approximation subspace tracking. *Signal Processing, IEEE Transactions on*, 43(1):95–107, 1995.

[4] Jimeng Sun, Spiros Papadimitriou, and Christos Faloutsos. Distributed pattern discovery in multiple streams. In *Advances in Knowledge Discovery and Data Mining*, pages 713–718. Springer, 2006.

[5] Odysseas Papapetrou, Minos Garofalakis, and Antonios Deligiannakis. Sketch-based querying of distributed sliding-window data streams. *Proceedings of the VLDB Endowment*, 5(10):992–1003, 2012.

[6] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate query processing using wavelets. *The VLDB Journal—The International Journal on Very Large Data Bases*, 10(2-3):199–223, 2001.

[7] Yannis E Ioannidis and Viswanath Poosala. Histogram-based approximation of set-valued query-answers. In *VLDB*, volume 99, pages 174–185, 1999.

[8] Phillip B Gibbons and Yossi Matias. New sampling-based summary statistics for improving approximate query answers. In *ACM SIGMOD Record*, volume 27, pages 331–342. ACM, 1998.

[9] Sameer Agarwal, Henry Milner, Ariel Kleiner, Ameet Talwalkar, Michael Jordan, Samuel Madden, Barzan Mozafari, and Ion Stoica. Knowing when you're wrong: Building fast and reliable approximate query processing systems.

[10] N Alon, P Gibbons, Y Matias, and M Szegedy. Tracking joins and self joins in limited storage. In *ACM PODS Conference*, 1999.

[11] Alin Dobra, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing complex aggregate queries over data streams. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 61–72. ACM, 2002.

[12] Alin Dobra, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Sketch-based multi-query processing over data streams. In *Advances in Database Technology-EDBT 2004*, pages 551–568. Springer, 2004.

[13] Sumit Ganguly, Minos Garofalakis, and Rajeev Rastogi. Processing data-stream join aggregates using skimmed sketches. In *Advances in Database Technology-EDBT 2004*, pages 569–586. Springer, 2004.

[14] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, 31(6):1794–1813, 2002.

[15] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Automata, Languages and Programming*, pages 693–703. Springer, 2002.

[16] Graham Cormode and S Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[17] Charu C Aggarwal, Jiawei Han, Jianyong Wang, and Philip S Yu. A framework for clustering evolving data streams. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 81–92. VLDB Endowment, 2003.

[18] Robert Schweller, Ashish Gupta, Elliot Parsons, and Yan Chen. Reversible sketches for efficient and accurate change detection over network data streams. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 207–212. ACM, 2004.

[19] Charu C Aggarwal, Jiawei Han, Jianyong Wang, and Philip S Yu. On demand classification of data streams. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 503–508. ACM, 2004.

[20] Christos Boutsidis, Dan Garber, Zohar Karnin, and Edo Liberty. Online principal component analysis.

[21] Matthew Brand. Fast low-rank modifications of the thin singular value decomposition. *Linear algebra and its applications*, 415(1):20–30, 2006.

[22] Spiros Papadimitriou, Jimeng Sun, and Christos Faloutsos. Streaming pattern discovery in multiple time-series. In *Proceedings of the 31st international conference on Very large data bases*, pages 697–708. VLDB Endowment, 2005.

[23] Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 358–369. VLDB Endowment, 2002.

[24] Rudolf Kulhavỳ. Restricted exponential forgetting in real-time identification. *Automatica*, 23(5):589–600, 1987.

[25] Charu C Aggarwal. On futuristic query processing in data streams. In *Advances in Database Technology-EDBT 2006*, pages 41–58. Springer, 2006.

[26] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[27] Pavlos S Efraimidis and Paul G Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters*, 97(5):181–185, 2006.

[28] Sudipto Guha, Nick Koudas, and Kyuseok Shim. Data-streams and histograms. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 471–475. ACM, 2001.

[29] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 346–357. VLDB Endowment, 2002.

[30] Richard M Karp, Scott Shenker, and Christos H Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems (TODS)*, 28(1):51–55, 2003.

[31] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Querying and mining data streams: you only get one look a tutorial. In *SIGMOD Conference*, volume 635, 2002.

[32] Graham Cormode and Minos Garofalakis. Sketching probabilistic data streams. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 281–292. ACM, 2007.

[33] Michael W Marcellin. *JPEG2000 Image Compression Fundamentals, Standards and Practice: Image Compression Fundamentals, Standards, and Practice*, volume 1. springer, 2002.

[34] K Onthriar, Kok-Keong Loo, and Z Xue. Performance comparison of emerging dirac video codec with h. 264/av. In *Digital Telecommunications,, 2006. ICDT'06. International Conference on*, pages 22–22. IEEE, 2006.

[35] Anna C Gilbert, Yannis Kotidis, S Muthukrishnan, and Martin Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *VLDB*, volume 1, pages 79–88, 2001.

[36] Graham Cormode, Minos Garofalakis, and Dimitris Sacharidis. Fast approximate wavelet tracking on streams. In *Advances in Database Technology-EDBT 2006*, pages 4–22. Springer, 2006.

[37] Volker Strumpen, Henry Hoffmann, and Anant Agarwal. A stream algorithm for the svd. 2003.

[38] Charu C Aggarwal, Jiawei Han, Jianyong Wang, and Philip S Yu. A framework for projected clustering of high dimensional data streams. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 852–863. VLDB Endowment, 2004.

[39] Feng Cao, Martin Ester, Weining Qian, and Aoying Zhou. Density-based clustering over an evolving data stream with noise. In *SDM*, volume 6, pages 326–337. SIAM, 2006.

[40] Moses Charikar, Liadan O'Callaghan, and Rina Panigrahy. Better streaming algorithms for clustering problems. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 30–39. ACM, 2003.

[41] Haixun Wang, Wei Fan, Philip S Yu, and Jiawei Han. Mining concept-drifting data streams using ensemble classifiers. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 226–235. ACM, 2003.

[42] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80. ACM, 2000.

[43] Graham Cormode and S Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. *ACM Transactions on Database Systems (TODS)*, 30 (1):249–278, 2005.

[44] Chris Giannella, Jiawei Han, Jian Pei, Xifeng Yan, and Philip S Yu. Mining frequent patterns in data streams at multiple time granularities. *Next generation data mining*, 212:191–212, 2003.

[45] Piotr Indyk, Nick Koudas, and S Muthukrishnan. Identifying representative trends in massive time series data sets using sketches. In *VLDB*, pages 363–372, 2000.

[46] Richard Cole, Dennis Shasha, and Xiaojian Zhao. Fast window correlations over uncooperative time series. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 743–749. ACM, 2005.

[47] Yixin Chen, Guozhu Dong, Jiawei Han, Benjamin W Wah, and Jianyong Wang. Multi-dimensional regression analysis of time-series data streams. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 323–334. VLDB Endowment, 2002.

[48] Yasushi Sakurai, Spiros Papadimitriou, and Christos Faloutsos. Braid: Stream mining through group lag correlations. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 599–610. ACM, 2005.

[49] Raman Arora, Andrew Cotter, Karen Livescu, and Nathan Srebro. Stochastic optimization for pca and pls. In *Allerton Conference*, pages 861–868. Citeseer, 2012.

[50] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2005.

[51] Apache Hadoop. Hadoop, 2009.

[52] Apache Cassandra. The apache software foundation. *URL: http://cassandra. apache. org/(visited on 01/05/2013)*.

[53] Apache HBase. The apache hadoop project.

[54] Apache Foundation. ZooKeeper: A Distributed and fault-tolerant highly reliable distributed coordination environment. https://zookeeper.apache.com/.

[55] David Heinemeier Hansson et al. Ruby on rails. *Website. Projektseite: http://www. rubyonrails. org*, 2009.

[56] Yukio Matsumoto and K Ishituka. Ruby programming language, 2002.

[57] Apache Mahout. Scalable machine learning and data mining, 2012.

[58] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[59] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31 (5):532–533, 1988.

[60] Neil J Gunther, Shanti Subramanyam, and Stefan Parvu. A methodology for optimizing multithreaded system scalability on multi-cores. *arXiv preprint arXiv:1105.4301*, 2011.